<table>
<thead>
<tr>
<th>ACTION</th>
<th>NAME</th>
<th>DATE</th>
<th>SIGNATURE</th>
</tr>
</thead>
<tbody>
<tr>
<td>WRITTEN BY</td>
<td>Jason W. Bacon</td>
<td>December 31, 2017</td>
<td></td>
</tr>
<tr>
<td>Initial HTcondor</td>
<td>Lars Olson</td>
<td>December 31, 2017</td>
<td></td>
</tr>
<tr>
<td>content</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Contents

1 Goals .................................................. 2

2 Self-Study Instructions .................................. 3
  2.1 Unix Self-Study Instructions .............................. 3
  2.2 Parallel Computing Self-Study Instructions .............. 3

I Research Computing ..................................... 4

3 Computational Science .................................. 5
  3.1 So, What is Computational Science? ....................... 5
  3.2 Not Just for Nerds Anymore ............................. 5
  3.3 The Ultimate Goal ........................................ 5
    3.3.1 The Computation Timeline ............................ 5
    3.3.2 Development Time ..................................... 6
    3.3.3 Deployment Time ....................................... 6
    3.3.4 Learning Time ........................................... 6
    3.3.5 Run Time ................................................ 6
  3.4 Common Methods Used in Computational Science .......... 6
    3.4.1 Numerical Analysis .................................... 6
    3.4.2 Computational Modeling ................................. 7
    3.4.3 Data Mining .............................................. 8
    3.4.4 Parameter Sweeps ....................................... 8
    3.4.5 Data Sifting .............................................. 9
    3.4.6 Monte Carlo .............................................. 9
    3.4.7 Everything Else ......................................... 10
  3.5 Self-test .............................................. 10
4 Where do I get the Software? 11
  4.1 But I Hate Programming... 11
  4.2 Buy It 11
  4.3 Download It 12
    4.3.1 How to Shoot Yourself in the Foot with Open Source Software 12
    4.3.2 How Not to Shoot Yourself in the Foot with Open Source Software 13
  4.4 The Realities of Software Support 14
  4.5 A Comprehensive List of Research Software 14
  4.6 Write It 15
  4.7 Self-test 16

5 Running Your Software 17
  5.1 Where do I Run It? 17
    5.1.1 Public Resources 17
    College Computer Labs 17
    College Clusters and Grids . 17
    XSEDE, Open Science Grid . 17
    Commercial Services . 17
    5.1.2 Using Your Own Computers 17
    Mac and other Unix 18
    Windows 18
    Virtual Machines 18
  5.2 Self-test 18

6 Using UWM Resources 19
  6.1 Obtaining Access 19
  6.2 Good Neighbor Guidelines 19
    6.2.1 Security 19
    6.2.2 Resource Use 20
  6.3 Connecting to UWM Resources 21
    6.3.1 Logging Into the Shell Environment 21
    6.3.2 File Transfers 21
    Mortimer File Transfers 22
  6.4 Cluster Data Storage 23
  6.5 Policies and Guidelines 24
  6.6 Using Available Software 25
  6.7 Code Development and Testing 26
  6.8 Off-campus Access 27
## Introduction to Unix

### 7.1 Aw, Man... I have to Learn a Another Operating System?  

### 7.2 What is Unix?  

#### 7.2.1 It’s All the Same...  

#### 7.2.2 Operating System, or Religion?  

#### 7.2.3 The Unix Standard API  

#### 7.2.4 Shake Out the Bugs  

#### 7.2.5 The Unix Standard UI  

#### 7.2.6 Freedom of Choice  

#### 7.2.7 Fast, Stable and Secure  

#### 7.2.8 Sharing Resources  

### 7.3 Self-test  

### 7.4 Unix User Interfaces  

#### 7.4.1 Graphical User Interfaces (GUIs)  

#### 7.4.2 X11 on Mac OS X  

#### 7.4.3 Command Line Interfaces (CLIs): Unix Shells  

#### 7.4.4 Terminals  

#### 7.4.5 Basic Shell Use  

#### 7.4.6 Self-test  

### 7.5 Still Need Windows? Don’t Panic!  

#### 7.5.1 Cygwin: It’s Worth a Shot  

#### 7.5.2 Unix and Windows Together?  

### 7.6 Logging In Remotely  

#### 7.6.1 Unix to Unix  

#### 7.6.2 Windows to Unix  

#### 7.6.3 Terminal Types  

#### 7.6.4 Self-test  

### 7.7 Unix Command Basics  

#### 7.7.1 Self-test  

### 7.8 Basic Shell Tools  

#### 7.8.1 Common Unix Shells  

#### 7.8.2 Command History  

#### 7.8.3 Auto-completion  

#### 7.8.4 Command-line Editing  

#### 7.8.5 Globbing (File Specifications)  

#### 7.8.6 Self-test  

### 7.9 Processes  

---

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Aw, Man... I have to Learn a Another Operating System?</td>
<td>30</td>
</tr>
<tr>
<td>What is Unix?</td>
<td>31</td>
</tr>
<tr>
<td>It’s All the Same...</td>
<td>31</td>
</tr>
<tr>
<td>Operating System, or Religion?</td>
<td>32</td>
</tr>
<tr>
<td>The Unix Standard API</td>
<td>32</td>
</tr>
<tr>
<td>Shake Out the Bugs</td>
<td>33</td>
</tr>
<tr>
<td>The Unix Standard UI</td>
<td>34</td>
</tr>
<tr>
<td>Freedom of Choice</td>
<td>34</td>
</tr>
<tr>
<td>Fast, Stable and Secure</td>
<td>35</td>
</tr>
<tr>
<td>Sharing Resources</td>
<td>35</td>
</tr>
<tr>
<td>Self-test</td>
<td>35</td>
</tr>
<tr>
<td>Unix User Interfaces</td>
<td>36</td>
</tr>
<tr>
<td>Graphical User Interfaces (GUIs)</td>
<td>36</td>
</tr>
<tr>
<td>X11 on Mac OS X</td>
<td>38</td>
</tr>
<tr>
<td>Command Line Interfaces (CLIs): Unix Shells</td>
<td>39</td>
</tr>
<tr>
<td>Terminals</td>
<td>40</td>
</tr>
<tr>
<td>Basic Shell Use</td>
<td>42</td>
</tr>
<tr>
<td>Self-test</td>
<td>43</td>
</tr>
<tr>
<td>Still Need Windows? Don’t Panic!</td>
<td>44</td>
</tr>
<tr>
<td>Cygwin: It’s Worth a Shot</td>
<td>44</td>
</tr>
<tr>
<td>Unix and Windows Together?</td>
<td>56</td>
</tr>
<tr>
<td>Logging In Remotely</td>
<td>56</td>
</tr>
<tr>
<td>Unix to Unix</td>
<td>57</td>
</tr>
<tr>
<td>Windows to Unix</td>
<td>58</td>
</tr>
<tr>
<td>Cygwin</td>
<td>58</td>
</tr>
<tr>
<td>PuTTY</td>
<td>58</td>
</tr>
<tr>
<td>Terminal Types</td>
<td>58</td>
</tr>
<tr>
<td>Self-test</td>
<td>59</td>
</tr>
<tr>
<td>Unix Command Basics</td>
<td>59</td>
</tr>
<tr>
<td>Self-test</td>
<td>61</td>
</tr>
<tr>
<td>Basic Shell Tools</td>
<td>61</td>
</tr>
<tr>
<td>Common Unix Shells</td>
<td>61</td>
</tr>
<tr>
<td>Command History</td>
<td>62</td>
</tr>
<tr>
<td>Auto-completion</td>
<td>62</td>
</tr>
<tr>
<td>Command-line Editing</td>
<td>62</td>
</tr>
<tr>
<td>Globbing (File Specifications)</td>
<td>63</td>
</tr>
<tr>
<td>Self-test</td>
<td>63</td>
</tr>
<tr>
<td>Processes</td>
<td>64</td>
</tr>
<tr>
<td>Section</td>
<td>Title</td>
</tr>
<tr>
<td>---------</td>
<td>-------</td>
</tr>
<tr>
<td>7.9.1</td>
<td>Self-test</td>
</tr>
<tr>
<td>7.10</td>
<td>The Unix File system</td>
</tr>
<tr>
<td>7.10.1</td>
<td>Unix Files</td>
</tr>
<tr>
<td></td>
<td>Text vs Binary Files</td>
</tr>
<tr>
<td></td>
<td>Unix vs. Windows Text Files</td>
</tr>
<tr>
<td>7.10.2</td>
<td>File system Organization</td>
</tr>
<tr>
<td></td>
<td>Basic Concepts</td>
</tr>
<tr>
<td></td>
<td>Full Path Names</td>
</tr>
<tr>
<td></td>
<td>Current Working Directory</td>
</tr>
<tr>
<td></td>
<td>Relative Path Names</td>
</tr>
<tr>
<td></td>
<td>Avoid Full Path Names</td>
</tr>
<tr>
<td></td>
<td>Special Directory Names</td>
</tr>
<tr>
<td>7.10.3</td>
<td>Ownership and Permissions</td>
</tr>
<tr>
<td></td>
<td>Overview</td>
</tr>
<tr>
<td></td>
<td>Viewing Permissions</td>
</tr>
<tr>
<td></td>
<td>Setting Permissions</td>
</tr>
<tr>
<td>7.10.4</td>
<td>Advanced: Unix File System Comparison</td>
</tr>
<tr>
<td>7.10.5</td>
<td>Advanced: Network File System</td>
</tr>
<tr>
<td>7.10.6</td>
<td>Self-test</td>
</tr>
<tr>
<td>7.11</td>
<td>Unix Commands and the Shell</td>
</tr>
<tr>
<td>7.11.1</td>
<td>Internal Commands</td>
</tr>
<tr>
<td>7.11.2</td>
<td>External Commands</td>
</tr>
<tr>
<td>7.11.3</td>
<td>Getting Help</td>
</tr>
<tr>
<td>7.11.4</td>
<td>A Basic Set of Unix Commands</td>
</tr>
<tr>
<td></td>
<td>File and Directory Management</td>
</tr>
<tr>
<td></td>
<td>Shell Internal Commands</td>
</tr>
<tr>
<td></td>
<td>Text File Processing</td>
</tr>
<tr>
<td></td>
<td>Text Editors</td>
</tr>
<tr>
<td></td>
<td>Networking</td>
</tr>
<tr>
<td></td>
<td>Identity and Access Management</td>
</tr>
<tr>
<td></td>
<td>Terminal Control</td>
</tr>
<tr>
<td>7.11.5</td>
<td>Self-test</td>
</tr>
<tr>
<td>7.12</td>
<td>Unix Command Quick Reference</td>
</tr>
<tr>
<td>7.13</td>
<td>POSIX and Extensions</td>
</tr>
<tr>
<td>7.14</td>
<td>File Transfer</td>
</tr>
<tr>
<td>7.14.1</td>
<td>File Transfers from Unix</td>
</tr>
<tr>
<td>7.14.2</td>
<td>File Transfer from Windows without Cygwin</td>
</tr>
<tr>
<td>7.14.3</td>
<td>Self-test</td>
</tr>
<tr>
<td>7.15</td>
<td>Environment Variables</td>
</tr>
</tbody>
</table>
7.15.1 Self-test ................................................................. 88
7.16 Shell Variables ......................................................... 89
  7.16.1 Self-test ............................................................. 89
7.17 More Shell Tools ....................................................... 89
  7.17.1 Redirection and Pipes ............................................. 89
    Device Independence .................................................. 89
    Redirection ........................................................... 90
    Special Files in /dev ............................................... 91
    Pipes ................................................................. 92
  7.17.2 Output Capture .................................................. 93
  7.17.3 Self-test .......................................................... 93
7.18 Process Control ..................................................... 94
  7.18.1 External Commands .............................................. 94
  7.18.2 Special Key Combinations ..................................... 95
  7.18.3 Internal Shell Commands and Symbols ....................... 95
  7.18.4 Self-test ........................................................ 96
7.19 Running a Desktop Unix System .................................. 97
7.20 Software Management ............................................... 98
  7.20.1 The Stone Age vs. Today ....................................... 98
  7.20.2 Lots and Lots of Package Managers ........................... 99
  7.20.3 Choosing Your Unix the Smart Way ......................... 100
  7.20.4 Self-test ........................................................ 101
7.21 Advanced: Enabling Remote Graphics ......................... 102
  7.21.1 Configuration Steps Common to all Operating Systems .... 102
  7.21.2 Graphical Programs on Windows with Cygwin ................ 103
    Installation ......................................................... 103
    Configuration ..................................................... 103
    Start-up .......................................................... 103
7.22 Advanced: Running Multiple Operating Systems .............. 103
7.23 Where to Learn More ............................................... 104

8 Unix Shell Scripting .................................................. 106
  8.1 What is a Shell Script? ........................................... 106
    8.1.1 Self-test ..................................................... 106
  8.2 Scripts vs Programs ............................................... 106
    8.2.1 Self-test ..................................................... 107
  8.3 Why Write Shell Scripts? ........................................ 107
    8.3.1 Efficiency and Accuracy ................................. 107
      Self-test ..................................................... 107
<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>8.3.2 Documentation</td>
<td>108</td>
</tr>
<tr>
<td>8.3.3 Why Unix Shell Scripts?</td>
<td>108</td>
</tr>
<tr>
<td>8.3.4 Self-test</td>
<td>108</td>
</tr>
<tr>
<td>8.4 Which Shell?</td>
<td>109</td>
</tr>
<tr>
<td>8.4.1 Common Shells</td>
<td>109</td>
</tr>
<tr>
<td>8.4.2 Self-test</td>
<td>109</td>
</tr>
<tr>
<td>8.5 Writing and Running Shell Scripts</td>
<td>109</td>
</tr>
<tr>
<td>8.5.1 Self-test</td>
<td>113</td>
</tr>
<tr>
<td>8.6 Shell Start-up Scripts</td>
<td>113</td>
</tr>
<tr>
<td>8.6.1 Self-test</td>
<td>114</td>
</tr>
<tr>
<td>8.7 Sourcing Scripts</td>
<td>114</td>
</tr>
<tr>
<td>8.7.1 Self-test</td>
<td>115</td>
</tr>
<tr>
<td>8.8 Scripting Constructs</td>
<td>115</td>
</tr>
<tr>
<td>8.9 Strings</td>
<td>115</td>
</tr>
<tr>
<td>8.10 Output</td>
<td>115</td>
</tr>
<tr>
<td>8.10.1 Self-test</td>
<td>117</td>
</tr>
<tr>
<td>8.11 Shell and Environment Variables</td>
<td>117</td>
</tr>
<tr>
<td>8.11.1 Assignment Statements</td>
<td>117</td>
</tr>
<tr>
<td>8.11.2 Variable References</td>
<td>119</td>
</tr>
<tr>
<td>8.11.3 Using Variables for Code Quality</td>
<td>120</td>
</tr>
<tr>
<td>8.11.4 Output Capture</td>
<td>121</td>
</tr>
<tr>
<td>8.11.5 Self-test</td>
<td>121</td>
</tr>
<tr>
<td>8.12 Hard and Soft Quotes</td>
<td>121</td>
</tr>
<tr>
<td>8.12.1 Self-test</td>
<td>122</td>
</tr>
<tr>
<td>8.13 User Input</td>
<td>122</td>
</tr>
<tr>
<td>8.13.1 Self-test</td>
<td>123</td>
</tr>
<tr>
<td>8.14 Conditional Execution</td>
<td>123</td>
</tr>
<tr>
<td>8.14.1 Command Exit Status</td>
<td>123</td>
</tr>
<tr>
<td>8.14.2 If-then-else Statements</td>
<td>124</td>
</tr>
<tr>
<td>Bourne Shell Family</td>
<td>124</td>
</tr>
<tr>
<td>C shell Family</td>
<td>128</td>
</tr>
<tr>
<td>8.14.3 Conditional Operators</td>
<td>129</td>
</tr>
<tr>
<td>8.14.4 Case and Switch Statements</td>
<td>131</td>
</tr>
<tr>
<td>8.14.5 Self-test</td>
<td>133</td>
</tr>
<tr>
<td>8.15 Loops</td>
<td>134</td>
</tr>
<tr>
<td>8.15.1 For and Foreach</td>
<td>134</td>
</tr>
<tr>
<td>8.15.2 While Loops</td>
<td>135</td>
</tr>
</tbody>
</table>
12 Job Scheduling with SLURM

12.1 The SLURM Scheduler

12.1.1 Cluster Status

   The Ganglia Resource Monitor

12.1.2 Job Status

12.1.3 Using top

12.1.4 Job Submission

   Submission Scripts

   Definitions

   Common Flags

   SLURM Resource Requirements

   Batch Serial Jobs

   Interactive Jobs

   Batch Parallel Jobs (Job Arrays)

   Multi-core Jobs

   MPI Multi-core Jobs

   Environment Variables

12.1.5 Terminating a Job

12.1.6 Terminating Stray Processes

12.1.7 Viewing Output of Active Jobs

12.1.8 Checking Job Stats with sacct

12.1.9 Checking Status of Running Jobs with scontrol

12.1.10 Job Sequences

12.1.11 Self-test

12.2 Local Customizations

13 Job Scheduling with HTCondor

13.1 The HTCondor Resource Manager

13.1.1 A Unique Approach to Resource Management

13.1.2 HTCondor Terminology

13.1.3 Pool Status

13.1.4 Job Submission

   Submit Description Files

   Common Features

   HTCondor Resource Requirements

   Batch Serial Jobs

   Batch Parallel Jobs (Job Clusters)

   MPI (Multi-core) Jobs

   A Submit Description File Template
13.1.5 Job Status ................................................................. 189
13.1.6 Terminating a Job or Cluster ........................................ 189
13.1.7 Job Sequences ......................................................... 190
13.1.8 Self-test ................................................................. 190
13.2 Local Customizations .................................................. 191

14 Job Scheduling with PBS (TORQUE) .................................. 192
14.1 The PBS Scheduler ..................................................... 192
   14.1.1 Cluster Status .................................................... 192
      The Ganglia Resource Monitor ..................................... 193
   14.1.2 Job Status .......................................................... 193
   14.1.3 Using top .......................................................... 194
   14.1.4 Job Submission .................................................... 194
      Submission Scripts ................................................... 194
      Common Flags ....................................................... 196
      PBS Resource Requirements ...................................... 196
      Batch Serial Jobs .................................................... 197
      Batch Parallel Jobs (Job Arrays) ................................. 197
      MPI (Multi-core) Jobs .............................................. 198
      Environment Variables ............................................ 198
      A Submit Script Template ......................................... 199
   14.1.5 Terminating a Job ................................................ 201
   14.1.6 Viewing Output of Active Jobs .................................. 201
   14.1.7 Self-test ........................................................... 202

15 Job Scheduling with LSF ................................................ 203
15.1 The LSF Scheduler ..................................................... 203
   15.1.1 Submitting Jobs .................................................. 203
      Using bsub without a script ....................................... 203
      Using a Submission Script ....................................... 203
   15.1.2 Job Types .......................................................... 205
      Batch Serial Jobs .................................................... 205
      Interactive Jobs ..................................................... 205
      Waiting for Jobs to Complete .................................... 205
      Batch Parallel Jobs ................................................ 205
      MPI Jobs ............................................................... 206
   15.1.3 Job Control and Monitoring ...................................... 207
      Listing jobs .......................................................... 207
      Checking Progress .................................................. 207
Using top ................................................................. 208
Terminating Jobs ..................................................... 208
Email Notification ..................................................... 208

15.2 Good Neighbor Scheduling Strategies ......................... 208
   15.2.1 Job Distribution ............................................. 208

15.3 More Information .................................................. 209

III Programming for Performance ................................. 210

16 Software Development ............................................ 211
   16.1 Programming is NOT Inherently Painful ....................... 211
   16.2 Operating System Selection .................................... 211
   16.3 Language Types .................................................. 212
      16.3.1 Machine Language ......................................... 212
      16.3.2 Assembly Language ........................................ 213
      16.3.3 High Level Languages (HLLs) ............................... 214
         Overview ....................................................... 214
      16.3.4 Self-test .................................................... 214
   16.4 Language Selection .............................................. 214
      16.4.1 Decision Factors ............................................ 214
      16.4.2 Language or Religion? ..................................... 215
         Language and Operating System Evangelists .................. 215
         The Rationalizations Behind Language Fever ................. 215
         The Rational Approach ....................................... 215
         Object-Oriented Design (OOD) for Beginners ............... 216
      16.4.3 History of Programming Languages ....................... 218
      16.4.4 Compiled vs Interpreted Languages ....................... 219
         Overview ....................................................... 219
         Not Everything in Interpreted Languages is Interpreted .... 220
         Using a Compiled Language Doesn’t Guarantee a Fast Program 220
         Think About the Future ....................................... 221
         Compiled Languages are Not That Hard to Use ............ 221
         Memory Use ................................................... 222
      16.4.5 Portable Software ........................................... 222
      16.4.6 Common Compiled Languages .............................. 222
         C/C++ ......................................................... 222
         Fortran ....................................................... 222
         Conclusion ................................................... 223
      16.4.7 Common Interpreted Languages ............................ 223
MATLAB and Octave .................................................. 223
A Real Example for Comparison .................................. 226
R ................................................................. 230
Perl ............................................................. 230
Python ........................................................... 230
Mixing Interpreted and Compiled Languages ................. 230

16.4.8 Summary ..................................................... 231
16.4.9 Self-test ..................................................... 231

16.5 Engineering Product Life Cycle ............................. 232
16.5.1 Specification .............................................. 232
16.5.2 Design .................................................... 232
16.5.3 Implementation ........................................... 232
16.5.4 Testing .................................................... 233
16.5.5 Self-test ................................................... 233

17 Data Representation ............................................ 235
17.1 What Everyone Should Know About Data Representation .. 235
17.2 Orders of Magnitude ......................................... 235
17.3 Numeric Limitations of Computers .......................... 235
17.4 Fixed Point Number Systems ................................ 236
17.5 Modular Number Systems ................................... 237
17.6 Binary Data Representation .................................. 237
17.7 The Arabic Numeral System .................................. 238
17.8 Binary Fixed Point ............................................ 238
  17.8.1 Limitations of Binary ................................... 238
17.9 Unsigned Binary Integers ..................................... 238
17.10 Two’s Complement .......................................... 239
  17.10.1 Format ................................................... 239
  17.10.2 Negation .................................................. 239
  17.10.3 Addition and Subtraction ................................. 240
  17.10.4 Range ................................................... 240
  17.10.5 Comparison .............................................. 240
  17.10.6 Overflow Detection ...................................... 241
  17.10.7 Extension and Reduction ................................. 241
17.11 Floating Point ................................................ 241
  17.11.1 The Basics ................................................ 241
  17.11.2 A Simple Floating Point Format ......................... 242
  17.11.3 Overflow and Underflow ................................. 243
  17.11.4 Cost of Floating Point ................................... 243
17.12 Character Storage .................................................. 244
  17.12.1 ASCII ....................................................... 244
  17.12.2 ISO ......................................................... 245
  17.12.3 Unicode ..................................................... 245
  17.13 Self-test ......................................................... 245

18 Computer Hardware .................................................. 248
  18.1 Motivation ....................................................... 248
  18.2 What is a Computer? ............................................. 248
  18.3 Central Processing Unit ........................................ 248
  18.4 Non-volatile Auxiliary Storage: Disk, Tape and Flash ...... 248
  18.5 Electronic Memory ............................................... 249
     18.5.1 RAM and ROM ............................................ 249
     18.5.2 Memory Structure ....................................... 249
     18.5.3 Virtual Memory ......................................... 249
  18.6 Computer Word Size ............................................ 250
  18.7 Self-test ......................................................... 250

19 Introduction to Programming ....................................... 251
  19.1 C and C++ ...................................................... 251
     19.1.1 Self-test ................................................ 251
  19.2 Not Your Grandfather’s Fortran ................................ 252
     19.2.1 Self-test ................................................ 252
  19.3 High-level Program Structure .................................. 252
     19.3.1 C ......................................................... 252
     19.3.2 Fortran ................................................. 253
     19.3.3 Self-test ............................................... 254
  19.4 The Compilation Process ........................................ 255
     19.4.1 Compilation Stages ...................................... 255
     19.4.2 Portability .............................................. 255
     19.4.3 Using Libraries ......................................... 256
     19.4.4 Self-test ............................................... 257
  19.5 Comments ........................................................ 258
     19.5.1 Block Comments ......................................... 258
     19.5.2 Line Comments .......................................... 259
     19.5.3 Self-test ............................................... 259
  19.6 Strings ......................................................... 259
19.6.1 Self-test ................................................................. 259
19.7 The C Preprocessor ..................................................... 260
  19.7.1 #include .......................................................... 260
  19.7.2 #define ........................................................ 261
  19.7.3 Self-test ........................................................ 261
19.8 The Basics of Variable Definitions .................................. 261
  19.8.1 Self-test ........................................................ 262
19.9 C Statements .......................................................... 262
  19.9.1 Self-test ........................................................ 263
19.10 Fortran Statements ................................................... 263
  19.10.1 Subroutine Calls ............................................... 263
  19.10.2 Assignment Statements ...................................... 263
  19.10.3 Flow-control Statements .................................... 264
  19.10.4 Self-test ........................................................ 264
19.11 Fortran Modules ...................................................... 264
  19.11.1 Self-test ........................................................ 265
19.12 Fortran Intrinsic Functions and the C Standard Libraries ... 265
  19.12.1 C ............................................................... 265
  19.12.2 Self-test ........................................................ 265
  19.12.3 Fortran ........................................................ 265
19.13 Code Quality .......................................................... 266
  19.13.1 Self-test ........................................................ 266
19.14 Performance .......................................................... 266
  19.14.1 Code Examples ............................................... 267
  19.14.2 Self-test ........................................................ 267

20 Data Types .............................................................. 268
  20.1 Internal Representation .......................................... 268
  20.2 Standard C Types .................................................. 268
    20.2.1 Self-test ...................................................... 270
  20.3 Standard Fortran Types .......................................... 270
    20.3.1 Self-test ...................................................... 270
  20.4 The Cost of Multiple Precision .................................. 271
    20.4.1 Self-test ...................................................... 272
  20.5 Variable Definitions .............................................. 272
    20.5.1 Initializers ................................................... 272
    20.5.2 Memory Maps ................................................ 273
    20.5.3 Self-test ...................................................... 273
  20.6 Constants ........................................................... 274
20.6.1 Literal Constants ................................................................. 274
       Self-test ........................................................................ 274
20.6.2 Named Constants ............................................................. 275
       Self-test ........................................................................ 277
20.7 Choosing the Best Data Type .................................................... 277
       20.7.1 Self-test .................................................................. 278
20.8 Math Operators and Expressions .............................................. 279
       20.8.1 Self-test .................................................................. 281
20.9 Mixing Data Types ................................................................. 281
       20.9.1 Implicit Data Conversions ......................................... 282
       20.9.2 Explicit Data Conversions ......................................... 283
       20.9.3 Self-test .................................................................. 284
20.10 Code Quality .................................................................... 284
20.11 Performance .................................................................... 284
       20.11.1 Code Examples ...................................................... 284
21 Basic Terminal Input/Output (I/O) .................................................. 285
    21.1 C Terminal Input/Output (I/O) ............................................. 285
       21.1.1 C Language Support ................................................ 285
       21.1.2 C Standard Streams ............................................... 285
       21.1.3 C Character I/O ....................................................... 286
       21.1.4 C String I/O ............................................................ 286
       21.1.5 C Formatted I/O ....................................................... 287
       21.1.6 Example of C input and output ................................... 288
       21.1.7 Using stderr with C ................................................. 289
       21.1.8 Self-test .................................................................. 289
    21.2 Fortran Terminal Input/Output (I/O) ....................................... 290
       21.2.1 Write and Print ......................................................... 290
       21.2.2 Read ...................................................................... 291
       21.2.3 One Statement, One Line .......................................... 291
       21.2.4 Standard Units ........................................................ 291
       21.2.5 Example of Fortran input and output ......................... 292
       21.2.6 Fortran Formatted I/O .............................................. 293
       21.2.7 Self-test .................................................................. 295
22 Conditional Execution 296

22.1 Motivation ................................................................. 296

22.2 Design vs. Implementation ............................................. 296

22.3 Boolean Expressions .................................................... 296

22.3.1 Boolean Constants ................................................. 296

22.3.2 Relational Expressions ............................................. 297

  Relational Operators .................................................. 297

  Boolean Operators .................................................... 298

22.3.3 Self-test ............................................................. 298

22.4 Flow Charts ............................................................ 298

22.5 If Statements .......................................................... 298

22.5.1 Minimal If Statements ............................................ 298

22.5.2 If with Multiple Statements ..................................... 299

22.5.3 Else ................................................................. 299

22.5.4 If-then-elseif-...-elseif-else-endif ............................ 300

22.5.5 Nesting If Statements ............................................ 301

22.5.6 Self-test ............................................................. 301

22.6 Switch, Select-Case .................................................... 302

22.6.1 Self-test ............................................................. 303

22.7 Code Quality .......................................................... 303

22.8 Performance ........................................................... 304

22.8.1 Code Examples ..................................................... 304

23 Loops (Iterative Execution) 305

23.1 Motivation ............................................................. 305

23.2 Design vs. Implementation ............................................. 305

23.3 Anatomy of a Loop .................................................... 305

23.3.1 Self-test ............................................................. 305

23.4 While: The Universal Loop .......................................... 305

23.4.1 Self-test ............................................................. 308

23.5 Fortran Fixed Do Loops ............................................... 309

23.5.1 Self-test ............................................................. 310

23.6 The C for Loop ........................................................ 310

23.6.1 Self-test ............................................................. 310

23.7 Fortran Unstructured Do Loops .................................... 311

23.7.1 Self-test ............................................................. 311

23.8 The C break Statement ............................................... 311

23.8.1 Self-test ............................................................. 312

23.9 The C do-while Loop ................................................ 312
24.12.3 Intent ......................................................... 337
24.13Top-down Programming and Stubs ........................... 338
24.14C Preprocessor Macros ....................................... 340
24.15Recursion ....................................................... 340
24.16Creating your own Libraries ................................ 340
24.17Self-test ......................................................... 340

25 Memory Addresses and Pointers .................................. 341

26 Arrays .......................................................... 343
26.1 Motivation ....................................................... 343
26.2 Design vs. Implementation .................................... 343
26.3 Array Basics ..................................................... 344
26.4 Dynamic Memory Allocation ................................... 347
26.5 Array Constants .................................................. 348
26.6 Static Array Initialization ...................................... 349
26.6.1 The Fortran DATA Statement ................................. 349
26.6.2 Look-up Tables ............................................... 349
26.7 Arrays as Arguments ............................................ 351
26.8 Allocate and Subprograms ...................................... 352
26.9 Sorting ........................................................... 354
26.10When to Use Arrays ............................................. 361
26.11Array Pitfalls ..................................................... 362
26.11.1 Address Calculation ......................................... 362
26.11.2 Out-of-bounds Errors ....................................... 362
26.11.3 Pinpointing Crashes with a Debugger ....................... 363
26.12Fortran 90 Vector Operations ................................... 364
26.12.1 Implicit Loops ............................................... 364
26.12.2 Where ......................................................... 365
26.13Code Quality ...................................................... 365
26.14The Memory Hierarchy ......................................... 366
26.14.1 Virtual Memory ............................................... 366
26.14.2 Cache ........................................................ 366
26.14.3 Memory Access Time ....................................... 366
26.15Performance ...................................................... 366
26.15.1 Code Examples ................................................. 366
26.16Self-test .......................................................... 366
29 Matrices

29.1 Motivation .................................................. 381

29.1.1 Tables .................................................. 381

29.1.2 Grids .................................................. 381

29.1.3 Bricks .................................................. 381

29.1.4 Systems of Equations ................................. 382

29.1.5 Matrix Representation .................................. 382

29.2 Multidimensional Arrays ................................. 382

29.3 Reading and Writing Matrices ............................ 384

29.4 Importance of Traversal Order ......................... 385

29.5 Multidimensional Arrays as Arguments ............... 386

29.6 Fortran Intrinsic Functions .............................. 387

29.7 Homework ................................................. 387

30 Software Performance ................................. 388

30.1 Motivation .................................................. 388

30.2 Analysis of Algorithms .................................... 388

30.2.1 Order of Operations: Big-O ......................... 388

30.2.2 Predicting Run Time .................................. 389

30.2.3 Determining Order .................................... 389

30.2.4 Common Algorithms .................................. 389

30.3 Performance Measurement .............................. 390

30.3.1 Timing Programs ..................................... 390

30.3.2 Profiling: Timing Subprograms ........................ 390

30.4 Homework ................................................. 391

31 Structures .................................................. 392

31.1 Motivation .................................................. 392

31.2 C Structures .............................................. 392

31.3 Fortran Structures ........................................ 392

31.4 Classes and Object-oriented Programming (OOP) ...... 392

31.5 Homework ................................................. 392

32 The Preprocessor ............................................. 393

32.1 Defined Constants ......................................... 393

32.2 Conditional Compilation .................................. 393

32.3 Homework ................................................. 393
33 Software Project Management .......................................................... 394
  33.1 Building with Make ................................................................... 394
    33.1.1 Building a Program ............................................................... 394
    33.1.2 Make Variables .................................................................. 395
    33.1.3 Special Targets ................................................................. 396
    33.1.4 Building Libraries ............................................................ 397
    33.1.5 Mixing Compilers .............................................................. 398
    33.1.6 Mixing C, C++, and Fortran: A Brief Overview .................... 398
      Data formats ........................................................................... 398
      Libraries ............................................................................... 398
  33.2 Software Deployment ................................................................ 400
  33.3 Version Control Systems .......................................................... 400
  33.4 Source Code Hosting ................................................................. 400
  33.5 Leave Dependencies to Package Managers ............................... 400
  33.6 Avoid Custom Build Systems .................................................... 401
  33.7 Follow the Filesystem Hierarchy Standard .................................. 401
  33.8 Package-friendly Software ......................................................... 402
    33.8.1 Modularity is the Key .......................................................... 402
    33.8.2 All You Need is a Simple Makefile .................................... 402
    33.8.3 Use Conventional Tools .................................................... 402
    33.8.4 Archiving Standards ........................................................ 402
    33.8.5 Respect the Environment ................................................... 402
    33.8.6 Install the Same Files Everywhere .................................... 402
    33.8.7 A Simple Example .............................................................. 402
      Debian Package ....................................................................... 402
      FreeBSD Port ......................................................................... 402
      MacPort ............................................................................... 402
      Pkgsrc Package ...................................................................... 402

IV Parallel Programming .................................................................. 403
34 Parallel Programming .................................................................... 404
  34.1 Serial vs. Parallel ..................................................................... 404
    34.1.1 Optimize the Serial Code First ............................................ 404
    34.1.2 Parallel Computing vs Parallel Programming ..................... 404
    34.1.3 Don’t Do Development on a Cluster or Grid ....................... 404
    34.1.4 Self-test ............................................................................ 405
  34.2 Scheduling Compilation .............................................................. 405
    34.2.1 Why Schedule Compilation? ............................................... 405
    34.2.2 Batch Serial and Batch Interactive .................................... 405
    34.2.3 Self-test ............................................................................ 406
  34.3 Further Reading ....................................................................... 406
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>35</td>
<td>Programming for HTC</td>
<td>407</td>
</tr>
<tr>
<td>35.1</td>
<td>Introduction</td>
<td>407</td>
</tr>
<tr>
<td>35.1.1</td>
<td>Self-test</td>
<td>407</td>
</tr>
<tr>
<td>35.2</td>
<td>Parallelize as a Last Resort</td>
<td>408</td>
</tr>
<tr>
<td>35.3</td>
<td>Common Uses for HTC</td>
<td>408</td>
</tr>
<tr>
<td>35.3.1</td>
<td>Monte Carlo Experiments</td>
<td>408</td>
</tr>
<tr>
<td></td>
<td>Software Performance Optimization</td>
<td>412</td>
</tr>
<tr>
<td></td>
<td>Calpi on a SLURM Cluster</td>
<td>415</td>
</tr>
<tr>
<td></td>
<td>Calpi on an HTCondor Grid</td>
<td>417</td>
</tr>
<tr>
<td>35.3.2</td>
<td>Parameter Sweeps and File Transfer</td>
<td>422</td>
</tr>
<tr>
<td></td>
<td>Calpi-file on a SLURM Cluster</td>
<td>423</td>
</tr>
<tr>
<td></td>
<td>Calpi-file on an HTCondor Grid</td>
<td>427</td>
</tr>
<tr>
<td>35.3.3</td>
<td>Self-test</td>
<td>429</td>
</tr>
<tr>
<td>36</td>
<td>Programming for HPC</td>
<td>430</td>
</tr>
<tr>
<td>36.1</td>
<td>Introduction</td>
<td>430</td>
</tr>
<tr>
<td>36.1.1</td>
<td>Self-test</td>
<td>430</td>
</tr>
<tr>
<td>36.2</td>
<td>Parallelize as a Last Resort</td>
<td>431</td>
</tr>
<tr>
<td>36.3</td>
<td>Common Uses for HPC</td>
<td>431</td>
</tr>
<tr>
<td>36.4</td>
<td>Real HPC Examples</td>
<td>431</td>
</tr>
<tr>
<td>36.5</td>
<td>HPC Programming Languages and Libraries</td>
<td>431</td>
</tr>
<tr>
<td>36.5.1</td>
<td>Self-test</td>
<td>432</td>
</tr>
<tr>
<td>36.6</td>
<td>Shared Memory Parallel Programming with OpenMP</td>
<td>432</td>
</tr>
<tr>
<td>36.6.1</td>
<td>OMP Parallel</td>
<td>433</td>
</tr>
<tr>
<td>36.6.2</td>
<td>OMP Loops</td>
<td>434</td>
</tr>
<tr>
<td>36.6.3</td>
<td>Shared and Private Variables</td>
<td>435</td>
</tr>
<tr>
<td>36.6.4</td>
<td>Critical and Atomic Sections</td>
<td>435</td>
</tr>
<tr>
<td>36.6.5</td>
<td>Self-test</td>
<td>437</td>
</tr>
<tr>
<td>36.7</td>
<td>Shared Memory Parallelism with POSIX Threads</td>
<td>437</td>
</tr>
<tr>
<td>36.8</td>
<td>Message Passing Interface (MPI)</td>
<td>438</td>
</tr>
<tr>
<td>36.8.1</td>
<td>Self-test</td>
<td>439</td>
</tr>
<tr>
<td>36.9</td>
<td>MPI vs Other Parallel Paradigms</td>
<td>439</td>
</tr>
<tr>
<td>36.9.1</td>
<td>Self-test</td>
<td>439</td>
</tr>
<tr>
<td>36.10</td>
<td>Structure of an MPI Job</td>
<td>439</td>
</tr>
<tr>
<td>36.11</td>
<td>A Simple MPI Program</td>
<td>440</td>
</tr>
<tr>
<td>36.11.1</td>
<td>Self-test</td>
<td>446</td>
</tr>
<tr>
<td>36.12</td>
<td>Best Practices with MPI</td>
<td>446</td>
</tr>
<tr>
<td>36.12.1</td>
<td>Self-test</td>
<td>447</td>
</tr>
<tr>
<td>36.13</td>
<td>Higher Level MPI Features</td>
<td>447</td>
</tr>
</tbody>
</table>
List of Figures

7.1 Sample of a Unix File system ......................................................... 66
7.2 Windows as a Guest under VirtualBox on a Mac Host ..................... 104
## List of Tables

<table>
<thead>
<tr>
<th>Table Number</th>
<th>Table Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.1</td>
<td>Computation Timeline</td>
<td>6</td>
</tr>
<tr>
<td>6.1</td>
<td>TCP Ports for Mortimer File Servers</td>
<td>22</td>
</tr>
<tr>
<td>6.2</td>
<td>Job Limits</td>
<td>25</td>
</tr>
<tr>
<td>6.3</td>
<td>Job Limits</td>
<td>25</td>
</tr>
<tr>
<td>7.1</td>
<td>Partial List of Unix Operating Systems</td>
<td>31</td>
</tr>
<tr>
<td>7.2</td>
<td>Default Key Bindings in some Shells</td>
<td>63</td>
</tr>
<tr>
<td>7.3</td>
<td>Globbing Symbols</td>
<td>63</td>
</tr>
<tr>
<td>7.4</td>
<td>Special Directory Symbols</td>
<td>69</td>
</tr>
<tr>
<td>7.5</td>
<td>Common hot keys in <code>more</code></td>
<td>75</td>
</tr>
<tr>
<td>7.6</td>
<td>Unix Commands</td>
<td>83</td>
</tr>
<tr>
<td>7.7</td>
<td>Common Extensions</td>
<td>84</td>
</tr>
<tr>
<td>7.8</td>
<td>Standard Streams</td>
<td>90</td>
</tr>
<tr>
<td>7.9</td>
<td>Redirection Operators</td>
<td>90</td>
</tr>
<tr>
<td>7.10</td>
<td>Pipe Operators</td>
<td>92</td>
</tr>
<tr>
<td>8.1</td>
<td>Conventional script filename extensions</td>
<td>110</td>
</tr>
<tr>
<td>8.2</td>
<td>Shell Start Up Scripts</td>
<td>113</td>
</tr>
<tr>
<td>8.3</td>
<td>Printf Format Specifiers</td>
<td>116</td>
</tr>
<tr>
<td>8.4</td>
<td>Special Character Sequences</td>
<td>116</td>
</tr>
<tr>
<td>8.5</td>
<td>Test Relational Operators</td>
<td>126</td>
</tr>
<tr>
<td>8.6</td>
<td></td>
<td>127</td>
</tr>
<tr>
<td>8.7</td>
<td>C Shell Relational Operators</td>
<td>129</td>
</tr>
<tr>
<td>8.8</td>
<td>Shell Conditional Operators</td>
<td>130</td>
</tr>
<tr>
<td>8.9</td>
<td></td>
<td>145</td>
</tr>
<tr>
<td>16.1</td>
<td>Example MIPS Instruction</td>
<td>213</td>
</tr>
<tr>
<td>16.2</td>
<td>Selection Sort of 100,000 Integers</td>
<td>220</td>
</tr>
<tr>
<td>17.1</td>
<td>Basic Quantities</td>
<td>235</td>
</tr>
<tr>
<td>17.2</td>
<td>Prefixes for larger quantities</td>
<td>236</td>
</tr>
<tr>
<td>Section</td>
<td></td>
<td></td>
</tr>
<tr>
<td>---------</td>
<td></td>
<td></td>
</tr>
<tr>
<td>17.3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>17.4</td>
<td></td>
<td></td>
</tr>
<tr>
<td>17.5</td>
<td></td>
<td></td>
</tr>
<tr>
<td>18.1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>20.1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>20.2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>20.3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>20.4</td>
<td></td>
<td></td>
</tr>
<tr>
<td>20.5</td>
<td></td>
<td></td>
</tr>
<tr>
<td>20.6</td>
<td></td>
<td></td>
</tr>
<tr>
<td>20.7</td>
<td></td>
<td></td>
</tr>
<tr>
<td>20.8</td>
<td></td>
<td></td>
</tr>
<tr>
<td>20.9</td>
<td></td>
<td></td>
</tr>
<tr>
<td>21.1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>21.2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>21.3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>21.4</td>
<td></td>
<td></td>
</tr>
<tr>
<td>21.5</td>
<td></td>
<td></td>
</tr>
<tr>
<td>22.1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>22.2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>22.3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>24.1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>24.2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>24.3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>26.1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>26.2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>29.1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>29.2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>29.3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>30.1</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Chapter 1

Goals

The overall goal of this user guide is to provide all the knowledge needed for researchers to get started with research computing at UWM. Different users have different needs and most will not need to read all of the chapters of this guide. The guide is divided into four parts, each of which is focused on the needs of typical types of researchers. You may only need the knowledge presented in one or two parts, or you may need it all!

After reading this document, you should know:

• How computers are commonly used in scientific research
• How to find and use available computing resources
• How to use Unix-compatible operating system environments, including BSD, Cygwin, Linux, and Mac OS X
• How to write portable shell scripts to automate the execution of your research tools on any operating system
• The types of parallel computing available today
• How to access the UWM research computing resources
• How to schedule typical jobs on clusters and grids
• Where to find more detailed information on all of the above

This document and other information regarding the UWM HPC Cluster Service can be found at:
http://www.uwm.edu/hpc/

If you have any questions about the content of this document, the website, or other cluster-related topics, please send an email to research-computing@uwm.edu.
Chapter 2

Self-Study Instructions

This guide is organized as a tutorial for users with little or no experience using the Unix command line or parallel computing resources.

2.1 Unix Self-Study Instructions

To begin learning the Unix environment, readers should do the following:

1. Get access to a Unix system if you don’t have it already. You will need this to practice running Unix commands and writing basic scripts. Apple Macintosh, BSD and Linux systems are all Unix compatible. If you are running Windows, you can quickly and easily add a Unix environment to it by installing Cygwin following the instructions in Section 7.5.1.

2. Thoroughly read Chapter 7 up to and including Section 7.10. The remaining sections can be covered later after gaining some hands-on experience. Do the self-test at the end of each section.

3. Thoroughly read Section 8.1 through Section 8.7. Do the Self-test at the end of each section.

2.2 Parallel Computing Self-Study Instructions

To begin learning the basics of parallel computing, readers should do the following:

1. Thoroughly read Chapter 6, Chapter 10 and Chapter 11.

2. Read Chapter 12, but don’t expect to understand it perfectly. Just familiarize yourself with the material so it will be easier to master during your first meeting with a facilitator.

3. If you plan to use the HTCondor pool, skim over Chapter 13.

The remaining chapters in the guide are more specialized and best tackled after becoming comfortable with basic HPC/HTC usage. They will not typically be discussed during the first meeting with your facilitator.
Part I

Research Computing
Chapter 3

Computational Science

3.1 So, What is Computational Science?

Nope, it’s not the study of computation. That would be computer science.

Computational science is any method of scientific exploration involving the use of computers. This may involve using computer models directly for experimentation or using computers to analyze data from experiments performed by other means.

3.2 Not Just for Nerds Anymore

Computation has been a core part of mathematics, physical science, and engineering research for decades. It is rapidly gaining popularity in other areas of research such as biology, psychology, economics, political science, and just about any other discipline you can think of.

This trend is due in part to the introduction of other technologies into these fields, such as rapid gene sequencers and imaging technology such as MRI (Magnetic Resonance Imaging). These new technologies generate vast amounts of data that require significant computing resources to store and process. Researchers in these fields often spend the majority of their time on the computer and only a small fraction in the lab. (If you don’t like computers, you may want to reconsider becoming a geneticist or MRI researcher.)

The trend is also due to computer technology itself facilitating the storage and use of vast amounts of data in all walks of life. The evolution of fast, cheap computer technology and the Internet has made it possible to archive detailed records of things like election results and sales records and make them easily available to almost anyone in the world. There are many researchers these days who don’t collect their own data, but simply use archives collected by others in the past.

3.3 The Ultimate Goal

3.3.1 The Computation Timeline

In doing computational science, our ultimate goal is to minimize the total time from the moment we decide what we want to do, to obtaining good quality results from the software we run.

Table 3.1 represents the timeline of computational science project, showing typical time requirements for developing the software, deploying (installing) the software, learning the software, and finally running the software.

Any one of these steps could end up taking the majority of your time, so we need to consider all of them when devising a strategy for achieving our goals. This text will discuss ways to minimize the time required for each step as well as potential trade-offs involved.
### 3.3.2 Development Time

Developing software inherently time-consuming. Large programs may require many thousands of man-hours to specify, design, implement, and test.

Fortunately, most researchers do not need to write their own software. There are many commercial and open source programs available to assist in a wide range of research methods.

### 3.3.3 Deployment Time

Deploying software can generally be quick and easy, but unfortunately, many researchers are not aware of efficient deployment methods and often end up wasting time or even failing to get their software installed.

### 3.3.4 Learning Time

Learning time is largely a matter of time, focus, and quality of documentation. All I can offer here is some advice: Do your homework, locate the best sources of documentation, and invest some time in studying it without distraction.

### 3.3.5 Run Time

Run time depends on many factors, such as the algorithms used by the program, the language used to implement it (compiled languages are many times faster than interpreted), and the hardware it runs on. Optimizing the software should always be done before throwing more hardware resources at it. When it is not feasible to make the program faster, one might consider using parallel computing resources. Doing so before optimizing the software would be an unwise and possibly unethical waste of resources. There is also a learning curve involved in using parallel resources that can be avoided by improving the program first.

### 3.4 Common Methods Used in Computational Science

Some of the most common computational science techniques are described below. However, this is not meant to suggest that they are the only ways to use computers for research. The only true limits are imposed by your own imagination.

#### 3.4.1 Numerical Analysis

Numerical analysis applies to many processes that occur in the real world that can be modeled by mathematical equations.

Unfortunately, many of these equations, while seemingly simple, cannot be solved directly by known analytical methods. The techniques taught in algebra, trigonometry, and calculus courses, while extremely powerful, apply to only a small fraction of the complex real world models that researchers encounter.

Numerical analysis takes care of the rest. For many models where it is difficult to find the answer, but easy to verify it, numerical analysis can be used to produce an approximation as precise as we want. Numerical techniques generally involve clever techniques to successively improve guesses at the answer or convert the model into a system of equations which can be solved directly. Both of these methods are tedious to perform by hand, but well-suited to a fast computer.

The classic first example often used in numerical analysis courses is Newton’s method for estimating the roots of an equation (where the graph of the equation crosses the x axis). The aim is to answer the question: "For what value(s) of x does f(x) = 0?". While finding the roots of some equations is not easy to do directly, it is generally easy to compute f(x) for any value of x and see how close it is to 0.
Now for the clever part: In most cases, a line tangent to \( f(x) \) will cross the x-axis at a point closer to a root than \( x \). This can be visualized by drawing a graph of an arbitrary function, and a series of tangent lines.

The equation for the tangent line is easily computed using \( x, f(x), \) and \( f'(x) \), the slope of the curve at \( f(x) \). The root of the tangent line (where the tangent line crosses the x axis) is then easily calculated using the equation for the line. This becomes the next "guess" for the root of the equation. With rare exceptions, this next guess for \( x \) will be closer to the root than the previous \( x \). We continue this process until \( f(x) \) is sufficiently close to 0 or until the difference between subsequent guesses at \( x \) is sufficiently small.

### 3.4.2 Computational Modeling

Computational models simulate real-world processes, following the relevant laws of math and physics.

Models might simulate the motion of individual molecules in a fluid or solid, or larger cells of fluid such as oil in an engine, water in the ocean, or air in the atmosphere. The weather forecasts we rely on (and complain about) are determined mainly through the use of large-scale fluid models utilizing measurements of temperature, pressure, humidity, and wind throughout the planet.

Models are also used to simulate traffic flow on roads and expressways that have not yet been built, usage patterns in buildings still in design phase, and population growth and collapse in remote ecosystems, to mention just a few more cases. Scientists, engineers and architects use models to find out things in advance that could otherwise get them in trouble, like "Will closing a lane during rush hour cause a traffic jam?" or "What will happen if we only put a bathroom on every other floor?".
3.4.3 Data Mining

The term "mining" traditionally refers to digging through vast amounts of earth to find small amounts of valuable minerals. The minerals are generally a very small fraction of the earth that’s removed, and it requires a lot of work to find and separate them.

There are also vast amounts of data stored on computers that contain small amounts of information of interest to a particular researcher. Data mining is the process of sifting through these data for "interesting" information.

There are many possible approaches to data mining. The ultimate goal is to have a computer search through huge archives or databases of information without human intervention and automatically identify items or patterns of interest. This is not always feasible, so a more practical goal is often to have the computer do as much as possible and simply minimize the human labor involved.

Depending on the type of information being searched, teaching a computer to identify truly interesting patterns can be fairly difficult and may require the use of artificial intelligence techniques. For example, a now-famous data mining experiment used to search hospital records for patterns initially reported that all of the maternity ward patients were women.

This underscores the fact that while computers are powerful tools that can do many things far faster and more accurately than humans, there are still many tasks that require human knowledge and reasoning.

3.4.4 Parameter Sweeps

Numerical analysis uses techniques to progressively improve guesses at the solution to a problem. Unfortunately, sometimes we can’t come up with a clever method of improving on our current guess and we simply have to test every possible answer until we find one that works.

A parameter sweep tests a range of possible answers to a question, until at least one correct answer is found. It is a brute-force approach to answering questions where more efficient methods are not known.

A classic example of a parameter sweep is the brute-force password hack. Passwords are stored in an encrypted form that cannot be directly converted back to the raw password. Since it is the raw password that must be entered in order to log into a computer, this effectively prevents unauthorized access even if the encrypted passwords are known. This is important since many passwords must be transmitted over networks in order to log into remote systems such as email servers. Hence, it is often difficult or impossible to prevent encrypted passwords from becoming known.

While it is practically impossible to decrypt a password, it is relatively straightforward to encrypt a guess and see if it matches the known encrypted form.

The main defense against this type of brute-force attack is forcing the attacker to try more guesses, i.e. maximizing the parameter space that must be swept. An 8-character password randomly consisting of both upper and lower case English letters, digits, and punctuation has \((26 + 26 + 10 + 32)^8 = 6.09 \times 10^{15}\) possible patterns (based on a US-English keyboard).

If a computer can encrypt and compare 100,000 guesses per second, it will take 1,932 years to sweep the entire parameter space. On average, it will take half that time to find one particular password with these qualities.
On the other hand, if the attacker knows that your password is a 10-letter English word with a mix of upper and lower case, then based on the size of the Oxford English dictionary (about 170,000 words), there are only about $170,000 \times 2^{10} = 174,080,000$ possible passwords. At 100,000 guesses per second, it would take the hacker's tools at most $1741$ seconds = 29 minutes to find your raw password.

For this reason, a password should never be any kind of derivative of a real word.

The worst kind of password, of course, is anything containing personal information. Many computer users think they're outsmarting hackers by putting a digit or two after their name to form a "secure" password. Hackers have a standard list of items commonly used by people just begging to get hacked, such as their name, birthday, pet's name, favorite color, etc. Most of this information is readily available online thanks to sites like Facebook. Checking every item in this list followed by every possible number from 1 to 999 (e.g. from azure0 to zebra999) will take only a fraction of a second on a modern computer.

### 3.4.5 Data Sifting

In some cases, there are just gobs and gobs of raw data to be sifted and checked for known or expected patterns. This is different from data mining in the sense that the human programmers know what to look for.

Examples of this type of computational research are well illustrated by the @home projects, such as Einstein@Home, which searches data from laser interferometer gravitational-wave observatory (LIGO) detectors for evidence of gravity waves. The LIGO detectors unfortunately don’t beep when they spot a gravity wave. Instead, they generate enormous amounts of data, most of which will not show any evidence of gravity waves, but nevertheless must be examined thoroughly. The Einstein@Home project uses massive numbers of personal computers around the world, each sweeping a small segment of the LIGO raw data.

### 3.4.6 Monte Carlo

Monte Carlo simulations, named after the gambling city in the French Riviera, utilize random numbers and simulation to piece together answers to scientific questions.

The method actually looks similar to a parameter sweep or data sift in that the same calculations are done on a large number of different inputs. However, the Monte Carlo method uses random inputs rather than a predetermined set of inputs. The random numbers generated are generally designed to be representative of the entire possible range, while being a fraction of the size.

For example, the average of a small, but truly random sample of a population is generally very close to the true average of the entire population. This mathematical fact makes many experiments possible in the humanities and social sciences, where sampling every member of a society is practically impossible.

A classic example of the Monte Carlo method is the estimation of the value of pi using a dart board.

Suppose we have a square dart board with a circle inscribed:

![Diagram](image)

The ratio of the area of the circle over the area of the square is $\pi/4$. 

---

---
If a bad enough darts player throws a large number of darts at the board, darts will end up randomly and uniformly distributed across the square board (and probably the surrounding wall). If and only if the darts are randomly and uniformly distributed, the ratio of the number of darts within the inscribed circle over the number of darts within the entire square board will then be close to $\pi/4$.

Statistically, the more darts are thrown, the closer the ratio will get to $\pi$.

Naturally, this process would take too long with a real dart board, so we might instead choose to simulate it on a computer. Most programming languages offer the ability to generate pseudo-random numbers within some fixed range with a uniform distribution. By randomly generating a sequence of $x$ and $y$ values with a uniform distribution, we can rapidly simulate throwing darts at a board and quickly develop an estimate for $\pi$.

### 3.4.7 Everything Else

The previous sections outline some of the commonly used methods in computational science.

Researchers can explore and understand these methods and also discover or invent new methods of their own for using computers in their research. The computational capacity of today’s computers is both vast and vastly underutilized. The possibilities for computational research are almost limitless and bounded only by the skills and imagination of researcher.

It is our hope that more researchers will simply begin to consider how computational methods might improve their research and then develop the skills and knowledge to tap into the vast and freely available hardware and software resources that are waiting to be utilized.

### 3.5 Self-test

1. Class discussion: What kind of research are you currently conducting, and how might computers be used to further your goals?
2. What is computational science?
3. Describe two reasons that computational science is growing so rapidly.
4. How do numerical analysis techniques work in general? (Describe the common approach they all use.)
5. Why are numerical analysis techniques so important to science and engineering?
6. Explain Newton’s method for finding the roots of a function. Use a graph to illustrate.
7. What is computational modeling? Describe two examples of processes commonly modeled on computers.
8. What is data mining? What is one of the major challenges in designing useful data mining software?
9. What is a parameter sweep? Describe one task that requires a parameter sweep. Is doing parameter sweeps desirable?
10. What is data sifting? Describe one real-world example that requires data sifting.
11. What is a Monte Carlo simulation? What do all Monte Carlo simulations have in common?
12. Describe one use of the Monte Carlo method to solve a problem.
13. Can you think of any computational science methods that do not fall into one of the categories described here?
Chapter 4

Where do I get the Software?

There are basically three ways to get the software you need:

1. Buy commercial software.
2. Download free software.
3. Write your own.

4.1 But I Hate Programming...

Did you ever get the feeling that you’re just wasting your time trying to write a program? Statistically, your gut feeling is probably correct. The fact is, most researchers today don’t need to do much programming, if any. There’s a vast number of both commercial and free software applications available to handle most of the computational needs of researchers, and more being developed all the time.

The percentage of researchers who need to write their own software is rapidly declining. Almost every day, another experimental program evolves into an organized, collaborative software development project, readily available to anyone in the world. Computational research is becoming more and more a matter of simply finding the right software for the job, installing it, and learning to use it.

In most cases, unless you’re an accomplished programmer doing cutting-edge computational science, your time will be better spent searching for software than writing your own.

4.2 Buy It

Commercial software packages are generally a good option for complex engineering computations such as fluid dynamics and finite element analysis.

Such software tends to be very costly to develop and therefore exists mainly for needs of wealthy industries such as automotive, aerospace, etc.

Commercial software is generally only available for a limited number of platforms, usually Windows, Mac, and a few select enterprise Linux distributions such as RHEL, CentOS, and SUSE. Furthermore, most commercial software is limited to specific versions of the supported operating systems. For example, some may not run on the latest version of Windows while others will only run on the latest version of Windows.

This can be a nuisance for those who use multiple commercial applications, which may not be available for the same platforms.

Most commercial software also requires managing licenses that typically limit use to a single computer or require managing a license server, which most IT professionals agree is worse than a root canal. License servers must remain available nearly 24/7, or
at least during all normal working hours, which means they have to deal with crazy people whenever the license server goes down, and routine maintenance has to be scheduled at times when they would rather be sleeping or on vacation. License management requires a significant amount of expertise and effort, which should be considered as part of the total cost of ownership (TCO) of the software.

All that said, where there’s a market for commercial software, the software often offers powerful capabilities not found elsewhere.

### 4.3 Download It

Fortunately for the vast population of underfunded researchers in most fields, there is a huge and growing collection of open source software available for research.

"Open source software" is software for which the source code is freely available. Source code (a collective noun, like "milk" and "honey") is the program in a human-readable language such as C, C++, Fortran, MATLAB, Python, R, etc. It must be compiled (translated to machine language by a program called a compiler) or interpreted by a program called an interpreter, in order to run on a computer.

The compilers and interpreters used to compile or run open source software are almost always open source as well, so there are usually no barriers to downloading, installing, and running open source applications, except your own computer skills.

Most open source software is written by people who use it for their own needs and share it freely with others for one or more of the following reasons:

1. They don’t see a market for their software, so it’s not worth trying to sell it.
2. They hate the busy work of running a business so much that they’d rather give their work away.
3. They’re just nice guys.

The quality of open source software varies from almost unusable, to superior to most commercial software. The only way to determine whether open source software will serve your needs is by exploring the available options. Things can change rapidly as well, so what you learned about software options a year ago may no longer apply.

The main advantages of open source over commercial software are:

1. It’s free.
2. There are no licenses to manage.
3. It will usually run on whatever hardware and operating system you prefer.

On the whole, open source software has come of age. It is now possible for most computer users to do their everyday work using exclusively open source operating systems such as FreeBSD and Linux and open source applications such as Firefox and LibreOffice.

### 4.3.1 How to Shoot Yourself in the Foot with Open Source Software

Many people fear open source software because they assume it is hard to install and learn.

Installation of open source software is actually far easier than commercial software installations when using a package manager such as Debian packages, FreeBSD ports, MacPorts, or Pkgsrc. Fear of open source software installations usually arises from a lack of awareness of package managers and subsequent unnecessary attempts to perform difficult and poorly-documented "caveman installations".

Example 4.1 describes a typical caveman installation for the R statistics package. Note that this example is relatively simple and well-documented compared to many.
Example 4.1 A Typical Caveman Installation

2.1 Simple compilation

First review the essential and useful tools and libraries in Essential and useful other programs under a Unix-alike, and install those you want or need. Ensure that the environment variable TMPDIR is either unset (and /tmp exists and can be written in and scripts can be executed from) or points to a valid temporary directory (one from which execution of scripts is allowed). Choose a directory to install the R tree (R is not just a binary, but has additional data sets, help files, font metrics etc). Let us call this place R_HOME. Untar the source code. This should create directories src, doc, and several more under a top-level directory: change to that top-level directory (At this point North American readers should consult Setting paper size.)

Issue the following commands:

```
./configure
make
```

(See Using make if your make is not called 'make'.)

Users of Debian-based 64-bit systems may need

```
./configure LIBnn=lib
make
```

Then check the built system works correctly by

```
make check
```

Failures are not necessarily problems as they might be caused by missing functionality,4 but you should look carefully at any reported discrepancies. (Some non-fatal errors are expected in locales that do not support Latin-1, in particular in true C locales and non-UTF-8 non-Western-European locales.) A failure in tests/ok-errors. R may indicate inadequate resource limits (see Running R). More comprehensive testing can be done by

```
make check-devel
```

or

```
make check-all
```

See file tests/README. If the configure and make commands execute successfully, a shell-script front-end called R will be created and copied to R_HOME/bin. You can link or copy this script to a place where users can invoke it, for example to /usr/local/bin/R. You could also copy the man page R.1 to a place where your man reader finds it, such as /usr/local/man/man1. If you want to install the complete R tree to, e.g., /usr/local/lib/R, see Installation. Note: you do not need to install R: you can run it from where it was built. You do not necessarily have to build R in the top-level source directory (say, TOP_SRCDIR). To build in BUILDDIR, run cd BUILDDIR TOP_SRCDIR/configure make and so on, as described further below. This has the advantage of always keeping your source tree clean and is particularly recommended when you work with a version of R from Subversion. (You may need GNU make to allow this, and you will need no spaces in the path to the build directory.)

Now rehash if necessary, type R, and read the R manuals and the R FAQ (files FAQ or doc/manual/R-FAQ.html, or http://CRAN.R-project.org/doc/FAQ/R-FAQ.html which always has the version for the latest release of R).

Before doing the above, however, one must also install dozens of other prerequisite packages, following a similar process for each one. This would include a compiler suite, GNU configure, possibly a make utility, and many math libraries on which R depends.

If you can follow the instructions and all goes well, you may be done with all this in a day or two. The biggest problem with caveman installations, however, is that they often fail, especially if you’re not using the exact same version of the exact same operating system as the developers. Since the developers of the software and all its prerequisites likely use a variety of operating systems, it’s very unlikely that you’ll get through any installation without running into problems that you’re probably not qualified to solve.

4.3.2 How Not to Shoot Yourself in the Foot with Open Source Software

Lucky for you, there are thousands of nerds like me in the world who think porting software is fun.

As a result, all of the pain described above can be avoided by simply choosing an operating system with a good package manager and learning how to use it.
FreeBSD ports, for example, makes it possible to install any one of over 20,000 open source software packages over the Internet, most often in a few seconds. Instead of following the instructions above from the R developers, a FreeBSD user would simply run:

```
pkg install R
```

This single command will automatically download and install R and all the necessary prerequisite packages required to run it. The developer of each FreeBSD port figured out the caveman installation instructions and pitfalls for FreeBSD and then automated the process in a standardized package called a "FreeBSD port" so that nobody else would ever have to duplicate that effort. Leveraging the work of port developers saves countless man-hours over the long run.

It’s possible to install FreeBSD, a complete desktop environment, and probably all the scientific software you need in about an hour on a typical PC with a fast Internet connection.

The experience would be similar for users of a Debian or Gentoo based Linux system. More details can be found in Section 7.20.

### 4.4 The Realities of Software Support

Many computer users fear open source software due to the lack of documentation and direct support from the vendor. These fears are largely unfounded, however.

Support for commonly used open source software is usually provided by the user community in the form of online forums and email lists, which are open to everyone, and easily searchable. For all but the most esoteric issues, answers to most of your questions are usually already posted on the Internet and easily found with a simple web search.

It’s true that nobody is obligated to help you, but in reality, even in a community where 90% of the forum participants are rude and arrogant, there are always a few people ready and willing to help. Those who do it well will politely point you to existing answers to your question, so you can learn to find your own answers in the future. Even the crabby, rude responses are often helpful, though, and you will learn to be grateful and understanding once you get over the blow to your ego.

Contrary to some users’ expectations, commercial software support does not guarantee answers to your questions either. Access to documentation is often restricted to registered customers who must log into a web site to view or retrieve it. Hence, a simple web search may not turn up any answers, because the search engines don’t have access to the documentation or discussions. Phone support often involves automated menu systems, spending time on hold, and difficulty finding a support person who can answer the question.

Software support can generally be summed up as follows:

- **Open Source:** You either find the answer to your question with a web search, post the question on a forum and get an answer in anywhere from a few minutes to a month, or, since you have the source code, you can fix it yourself. (The last option may or may not appeal to you.)

- **Commercial:** You either find the answer to your question with a web search, post the question on a forum and get an answer in anywhere from a few minutes to a month, call tech support and get an answer or get the company to fix the bug anywhere from a few minutes to a month, or, since you paid for it, you get to yell at somebody.

When determining whether to purchase a commercial software product, it’s best to simply decide whether the features are worth the added purchase cost and the costs and down time associated with license management. It may be that the commercial software offers capabilities or performance that are not currently available and any open source equivalent. If this will greatly improve your productivity, then it may justify the costs.

### 4.5 A Comprehensive List of Research Software

The number of commercial and free scientific software packages is too vast and too rapidly growing to be listed in any book. The best way to find out about software packages is by searching the WEB for strings such as "finite element software" or "statistical software".
Talking to colleagues can also be helpful, but keep in mind that they are most likely only knowledgeable about one or a few packages that they have been using, and may not even be aware of alternatives, especially newer ones.

Your WEB search will likely lead you to the many Wikipedia articles dedicated to providing an overview of software categories, such as List of finite element software packages, List of software for molecular mechanics modeling, and List of statistical packages.

Some software lists are embedded in other articles, such as Data mining.

A good way to get a quick overview of what’s available as established open source projects is looking at the listing of packages available in one of the top-tier package managers, such as Debian packages, FreeBSD ports, Gentoo Portage, MacPorts, or Pkgsrc.

For example, The FreeBSD ports website, http://www.freebsd.org/ports/index.html has a nicely organized and searchable listing of all currently available ports and packages.

Even if you don’t use FreeBSD, it’s an excellent resource to browse and search, just to see what software is available for Unix-compatible operating systems.

4.6 Write It

Those who have the programming skills and esoteric software needs may choose to write their own software.

For these people, choosing the right operating system and the right programming language are critical.

Writing software is time consuming (i.e. expensive), although not nearly as hard as most people make it for themselves. More on programming intelligently later...

The main goals when writing a program should always be to write as little of it as possible and to ensure that it will run on as many different computers as possible. Even if you cannot find a program that does exactly what you need, there are probably programs and libraries around that do most of what you need.

Your time will be much better spent finding established and well-tested software to incorporate into your programs, rather than writing everything yourself. For example, if your program involves typical matrix operations, there are many highly-efficient math libraries available that your program can use, such as BLAS, LAPACK, Eigen, Arpack, and METIS, just to name a few. Writing your own matrix multiplication routine would be an enormous waste of your own time and computer time, since the prewritten routines in one of the previously mentioned libraries are probably much faster than anything you would write yourself.

To use the BLAS library for matrix multiplication:

1. Install BLAS:
   - Debian packages: apt-get install blas
   - FreeBSD ports: pkg install blas
   - MacPorts: port install blas
   - Pkgsrc (CentOS, NetBSD and many others): cd math/blas && bmake install

2. Call the appropriate subroutine, such as dgemm(), within your program.

3. When compiling your program, tell the compiler to use the BLAS library:
   
   cc myprog.c -L/path/to/blas/lib -lblas

Most scientific software is developed on and for Unix compatible operating systems. This allows the software to be used on many different platforms, such as FreeBSD, Linux, Mac OS X, Solaris, etc. MS Windows is the only mainstream operating system that is not Unix compatible. Software written for MS Windows using proprietary tools such as Visual Studio or other commercial compilers, which cannot be used on other operating systems. On the other hand, software developed for Unix can usually be used on Windows with the help of a compatibility layer such as Cygwin. (See Section 7.5.1 for more details.)
Cygwin can also be used to develop Unix-compatible software on a Windows system. Most of the Unix development tools such as compilers and editors, as well as many other open source applications, can be easily installed via Cygwin’s package manager. Cygwin itself is free and can be downloaded and installed in about 10 minutes on a typical PC.

As far as programming languages are concerned, very small projects can often be done effectively using scripting languages such as MATLAB, Python, or R.

Larger projects requiring good performance should be done in a high-performance general-purpose compiled language such as C, C++, or Fortran.

Interpreted languages like MATLAB, Python, and R can do many common tasks easily and efficiently, but will never be able to do everything well. What they can do efficiently is limited to their built-in functions (which are written in compiled languages). Extensions written in an interpreted language will be slow. Extensions written in a compiled language are complicated and require mastering both languages, plus the interface between them. Many attempts at developing large software packages in interpreted languages end up being scrapped and restarted in a faster language.

Bad advice on choosing a language abounds in most professions. Many people choose languages for irrational reasons, such as finding the syntax pleasing, popularity among friends, etc.

A smart selection is based on objective measures such as portability (does it run on any operating system and processor type?), performance, price, etc. Table 16.2 provides a rough idea about the relative performance of many popular languages.

Detailed information about this table and language selection in general can be found in Section 16.4.

See Chapter 7 for more details about operating systems.

### 4.7 Self-test

1. Is it likely that you will need to write your own software for research? Why or why not?
2. What are the advantages of commercial software over open source?
3. What are the advantages of open source software over commercial?
4. Is open source software generally easier or harder to install than commercial software? Explain your answer.
5. Where can you find out about software that might serve your research needs?
6. What should be the primary goals of someone who does have to write their own software? How can the goals above be achieved?
Chapter 5

Running Your Software

5.1 Where do I Run It?

5.1.1 Public Resources

College Computer Labs

Most colleges and universities maintain computer labs with software to serve the needs of their students. Check with your instructors or department office to find out what’s available to you.

College Clusters and Grids

Some campuses may also have clusters and grids available for parallel computing. If you need to run large simulations, parameter sweeps, or Monte Carlo simulations, it may be possible to run hundreds at a time instead of one at a time on your PC or a stand-alone lab PC.

XSEDE, Open Science Grid

The National Science Foundation funds several very large clusters on campuses around the country for general use by researchers on other campuses.

Use of these resources is free for academic researchers. Small allocations of computing time are easily obtained, while larger allocations require a more extensive proposal.

Commercial Services

A number of commercial services are also available for those who have the ability to pay as they go.

The Milwaukee Institute offers computing time on a cluster for both academic and industry research.

Amazon EC2 allows researchers to create their own custom virtual clusters and pay for CPU time used.

5.1.2 Using Your Own Computers

Most scientific software is written for Unix-compatible operating systems. At the time of this writing, all mainstream operating systems except Microsoft Windows are Unix-compatible.
Mac and other Unix

If you have a Mac, you have a Unix system. There are many commercial software applications for Macs. There also also several package managers that can be used to easily install Unix-based open source software, including Fink, Homebrew, MacPorts, and Pkgsrc.

Hence, Mac is a good choice if you need to run both commercial software and Unix-based open source software, and don’t want to maintain multiple computers.

Many other Unix systems such as BSD and Linux will run on just about any PC (include a Mac).

For commercial software, you will generally need an enterprise Linux distribution such as Redhat, CentOS (essentially identical to Redhat), or SUSE Enterprise Linux. Most commercial software vendors will only support a few Linux distributions like these in addition to Mac and Windows. A few vendors support other platforms such as FreeBSD or Ubuntu, and other software can often be made to run on them, although the vendor will not provide any support.

If you are running mainly open source software, Debian, FreeBSD, Gentoo, and Ubuntu are generally better choices than the enterprise Linux systems, since they have vastly larger collections of packages to choose from and their open source packages tend to be more up-to-date than the Enterprise Linux systems. Enterprise systems intentionally run older versions of libraries and tools to maintain stability and binary compatibility with commercial applications for many years.

Windows

There are many commercial applications for Windows and some open source scientific software that has been ported to run directly under Windows. Most scientific software is more functional on a Unix-compatible system, however.

The Cygwin Unix-compatibility system allows most Unix software to run under Windows, and has an extensive package collection making it easy to install.

Cygwin and all the packages provided with it are open source and free. The Cygwin base system can be downloaded and installed in about 10 minutes. Installing additional packages is also quick and easy.

Generally, software will run slower under Cygwin than it would under a real Unix system on the same PC. Also note that the Cygwin package collection is not nearly as extensive as Debian packages, FreeBSD ports, and other top-tier package managers.

Nevertheless, Cygwin is a great option for quickly adding basic Unix functionality to a Windows PC, mainly because it is so easy to install and use.

See Section 7.5.1 for more details.

Virtual Machines

Another option for running both Windows and Unix software on the same machine is a virtual machine.

A virtual machine is essentially a software application that pretends to be a computer. This allows you to run an entire second operating system as a guest under your main operating system.

The best approach to this is usually running Windows as a guest under some sort of Unix system such as BSD or Linux, although it is possible to run Unix as a guest under Windows as well.

More information can be found in Section 7.22.

5.2 Self-test

1. Describe four different categories of hardware resources available for running research software.

2. What type of operating systems are used for most scientific software?

3. Which mainstream operating systems can be used to run most scientific software. Explain.

4. Describe two options for running scientific software on MS Windows computers.
Chapter 6

Using UWM Resources

Before You Begin
Before reading this chapter, you should have a basic understanding of Unix, remote login, and file transfers, which are covered in Section 7.6 and Section 7.14.

6.1 Obtaining Access

The research clusters, “Avi” and “Mortimer”, are available to faculty researchers and their assistants in participating schools and colleges. They are not currently available to students, except research assistants participating in faculty research.

The educational cluster, "Peregrine", is available to all UWM students for course work, research projects, or any other educational use.

The development server, "Sciprog" is available to all UWM affiliates for developing and testing their own code and testing third party applications.

To request access to any of these resources, visit http://www.uwm.edu/hpc/getting-started/.

To learn about other resources available for grant-related research, please visit the High Performance Computing Services website at http://www.uwm.edu/hpc/.

6.2 Good Neighbor Guidelines

As users of any shared resource, we have a responsibility to each other to be more conscientious than we need be with our own resources. With modern PCs, smart phones, and other personal devices, most people have little experience sharing computing resources with others.

This section is meant to draw your attention to issues specific to shared resources that you may not be accustomed to dealing with.

6.2.1 Security

Users sharing a computer system, or even a network, need to think about the impact of security breaches on other users as well as themselves. This will mean putting extra thought and effort into preventing others from accessing their account on the system.

One of the most important aspects of security is preventing others from discovering your password. There are several things every user must do to protect their password:

- Select a password that is hard to guess. NEVER use a password that contains your name, birthdate, or any other personal information.
• Never, under any circumstances, give your password to another person. If anyone asks for your password, be suspicious. Computer systems managers don’t need to know your password and nobody else has any business accessing your files without your help.

• Never write down your password or store it in an unencrypted file. If you are concerned about remembering your passwords, you can store them in a password vault such as keepassx. When you use a password vault, you need only remember one password for the rest of your life. All others can be stored safely in the vault so they can be looked up if you forget them. The keepassx application also allows you to cut and paste passwords without making them visible on your screen.

6.2.2 Resource Use

Each user of a shared resource such as a cluster has a responsibility to make their software run as efficiently as possible. CPU time on a cluster is an extremely expensive resource due not only to the cost of the hardware, but also the infrastructure for housing a huge array of delicate equipment and the labor costs required to maintain the equipment, install and upgrade software, train users, etc.

A medium-sized cluster will cost the community millions of dollars over its lifetime, so we must make every effort to ensure that the resources it provides are used productively. Programs that waste resources will prevent other programs from doing useful work when the cluster is under heavy load.

A common initial reaction when a program takes too long is to look for a faster computer. However, when software is inefficient, there are often far more gains to be made by optimizing the software than can be achieved by throwing more hardware at it. Switching from an interpreted language to a compiled language, finding better algorithms for expensive computations, or simply cleaning up program code can reduce run times by a factor of hundreds or even thousands.

Before looking for a faster computer, every programmer should learn how to profile their code. This means determining where the code is spending most of its time. This is important, because optimizing every part of a large program may be too time-consuming, and is usually unnecessary, since there are typically a few hot spots where a program uses 99% or more of its CPU time. By identifying and optimizing these hot spots, you will achieve near optimal performance for the program overall with minimal effort.
6.3 Connecting to UWM Resources

6.3.1 Logging Into the Shell Environment

All of our resources run Unix-compatible operating systems. Most tasks on the development servers and clusters are performed via the Unix command line interface (CLI), as described in Chapter 7.

To log into the development servers, or clusters for the purpose of running programs, you will need an ssh client program. Common ssh clients are discussed in Section 7.6. Use your ssh client to connect to one of the cluster login nodes:

- login.avi.hpc.uwm.edu
- login.mortimer.hpc.uwm.edu
- login.peregrine.hpc.uwm.edu

or to the development servers:

- sciprog.ceas.uwm.edu
- sciprog2.ceas.uwm.edu

Examples:

```
shell-prompt> ssh joe@sciprog.ceas.uwm.edu
shell-prompt> ssh chin@login.avi.hpc.uwm.edu
```

Note

For security reasons, these hosts are not directly accessible from off-campus. Users who wish to work from home will need to connect to the UWM network through another UWM computer with secure access or through the UWM VPN service, described in Section 6.8.

6.3.2 File Transfers

A cluster is a large group of separate computers, including login nodes, file servers, and compute nodes.

File transfers should not be performed through the cluster login nodes, login.*.hpc.uwm.edu. Doing so creates unnecessary load on the head nodes and impacts performance for other users.

All file transfers should be performed by connecting directly to the cluster file servers:

- data.avi.hpc.uwm.edu
- data.mortimer.hpc.uwm.edu
- data.peregrine.hpc.uwm.edu

You can use any sftp or scp capable transfer program, such as rsync, scp, WinSCP, or FileZilla. Connections should be configured to use SSH, SFTP, or SCP protocol.

Development servers are individual computers, not clusters. File transfers to and from the development servers therefore use the same host name as shell log-ins:

- sciprog.ceas.uwm.edu
- sciprog2.ceas.uwm.edu

For example, to transfer files to sciprog from any other Unix system:

```
shell-prompt> scp -r MyStudy sciprog.ceas.uwm.edu:
shell-prompt> sftp sciprog.ceas.uwm.edu:
shell-prompt> rsync -av --delete MyStudy sciprog.ceas.uwm.edu:
```

To transfer files to Avi from any other Unix system:

```
shell-prompt> scp -r MyStudy data.avi.hpc.uwm.edu:
shell-prompt> sftp data.avi.hpc.uwm.edu:
shell-prompt> rsync -av --delete MyStudy data.avi.hpc.uwm.edu:
```
Mortimer File Transfers

In order to parallelize file I/O as much as possible within the cluster, Mortimer uses many independent file servers. When you connect to Mortimer via SSH or using a file transfer program, you are actually connecting to a gateway, from which you can be routed directly to the login node or to any one of the independent file servers.

The default SSH port, 22, routes you to the login node. You can route your connection directly to any of the file servers by simply specifying an alternate TCP port instead of the default SSH port 22.

Your PC’s connection to the campus network is most likely capable of 1 gigabit per second. ( Wireless connections are much, much slower, however. )

Mortimer’s gateway connection to campus is capable of 10 gigbits per second. Hence, several simultaneous transfers to or from different file servers can fully utilize their 1 gigabit bandwidth, as they are sharing a 10 gigabit connection to the cluster. Simultaneous transfers to or from the same file server will share the 1 gigabit connection to that server, but this situation is very rare.

To connect to RAID server raid-XX (where XX is 01, 02, 03, etc.), use TCP port 220XX.

To connect to RAID server bigraid-XX (where XX is 01, 02, 03, etc.), use TCP port 221XX.

Table 6.1 lists the currently available Mortimer file servers and associated TCP ports.

If you’re not sure which file server your account uses, simply do a long listing of your ~/Data link (WITHOUT a trailing ‘/’):

```
Linux login.mortimer bacon ~ 401: ls -l ~/Data
lrwxrwxrwx. 1 bacon bacon 19 Dec 10 2014 /home/bacon/Data -> /raid-01/UITS/bacon/
```

Caution If you include a trailing ‘/’, you will list the contents of the ~/Data directory instead of showing what the link points to.

<table>
<thead>
<tr>
<th>Server</th>
<th>Port</th>
</tr>
</thead>
<tbody>
<tr>
<td>raid-01</td>
<td>22001</td>
</tr>
<tr>
<td>raid-02</td>
<td>22002</td>
</tr>
<tr>
<td>raid-03</td>
<td>22003</td>
</tr>
<tr>
<td>raid-04</td>
<td>22004</td>
</tr>
<tr>
<td>raid-05</td>
<td>22005</td>
</tr>
<tr>
<td>raid-06</td>
<td>22006</td>
</tr>
<tr>
<td>raid-07</td>
<td>22007</td>
</tr>
<tr>
<td>raid-08</td>
<td>22008</td>
</tr>
<tr>
<td>raid-09</td>
<td>22009</td>
</tr>
<tr>
<td>raid-10</td>
<td>22010</td>
</tr>
<tr>
<td>bigraid-01</td>
<td>22101</td>
</tr>
<tr>
<td>bigraid-02</td>
<td>22102</td>
</tr>
<tr>
<td>bigraid-03</td>
<td>22103</td>
</tr>
<tr>
<td>bigraid-04</td>
<td>22104</td>
</tr>
<tr>
<td>bigraid-05</td>
<td>22105</td>
</tr>
<tr>
<td>bigraid-06</td>
<td>22106</td>
</tr>
<tr>
<td>bigraid-07</td>
<td>22107</td>
</tr>
<tr>
<td>raid-768g-001/compute-768g-001</td>
<td>22044</td>
</tr>
</tbody>
</table>

Table 6.1: TCP Ports for Mortimer File Servers

Any SSH-based login or file transfer software can use this method. Programs such as WinSCP and FileZilla will allow you to enter the TCP port along with the hostname and login credentials when you configure a connection.
Command-line tools like ssh, scp, sftp, and rsync, use flag arguments to indicate a non-default port.

For example, to transfer files to raid-03 from any other Unix system:

```
shell-prompt> scp -r -p 22003 MyStudy data.mortimer.hpc.uwm.edu:
shell-prompt> sftp -o Port=22003 data.mortimer.hpc.uwm.edu
shell-prompt> rsync -av --delete -e 'ssh -p 22003’ MyStudy data.mortimer.hpc.uwm.edu:
```

To transfer files to bigraid-02 from any other Unix system:

```
shell-prompt> scp -r -p 22102 MyStudy data.mortimer.hpc.uwm.edu:
shell-prompt> sftp -o Port=22102 data.mortimer.hpc.uwm.edu
shell-prompt> rsync -av --delete -e 'ssh -p 22102’ MyStudy data.mortimer.hpc.uwm.edu:
```

### 6.4 Cluster Data Storage

This section assumes a basic knowledge of the Unix filesystem structure, which is discussed in Chapter 7. Some Unix books are also listed on the HPC Services website http://www4.uwm.edu/hpc/.

---

**Caution**

Cluster storage resources should be considered temporary storage, for use during active processing on the cluster. None of the data are currently backed up by an automated backup system. Given the large data capacity, it is not feasible to run regular backups due to the cost of backup storage, the impact that enormous backups would have on the campus network, and the impact they would have on jobs running on the cluster.

It is expected that users will generate data on the cluster, and then move the relevant results to their own computers for further analysis, publication, and archival. Any data generated on the cluster should be removed as soon as practical after processing on the cluster is complete.

---

You can view available storage by running the `df` command:

```
[bacon_a@hd1 ~]$ df
Filesystem  1K-blocks  Used  Available  Use%  Mounted on
/dev/mapper/KusuVolGroup00-ROOT 12092279  8182483  3295396  72%  /
/dev/mapper/KusuVolGroup00-HOME 114020510 35666056  72561072  33%  /home
/dev/mapper/KusuVolGroup00-DEPOT 10093008  5480384  4099805  58%  /depot
/dev/mapper/KusuVolGroup00-VAR  2031515  1026502   901794  54%  /var
/dev/sda2   102743   17759    79763  19%  /boot
/tmpfs      12337868  0  12337868    0%  /dev/shm
/dev/scd0  1556722   1556722    0  100%  /media/ANSSMyMultyPhysics
hd2-ib0:/share1   6832073984  65849344 6424546144  2%  /share1
hd2-ib0:/share2   7730003744 152248256  7191170688  3%  /share2
hd2-ib0:/share3   7964244416 5128371284 4237573632  68%  /share3
hd2-ib0:/share4   7730003744  181888  7343237056  1%  /share4
hd2-ib0:/share5   7730003744  247488  7343171456  1%  /share5
hd2-ib0:/share6   7964244416  637026400  6928918528  9%  /share6
hd2-ib0:/share7   7730003744  180800  7343238144  1%  /share7
hd2-ib0:/share8   7730003744  7142624  7335276320  1%  /share8
hd2-ib0:/share9   7964244416  1212928  7564732000  1%  /share9
hd2-ib0:/sharedapps  7730003744  60430784  7282988160  1%  /sharedapps
```

The `df` command shows all filesystems available to the computer on which `df` is run, and where they reside in the Unix directory structure.
Some directories, such as /home, /usr, and all the directories beginning with /share, are shared by all cluster nodes over the cluster’s local network. Hence, any files and directories placed in these directories by one node are immediately and directly accessible from all other nodes.

Other directories, such as /var, are only accessible to the head node. Each of the other nodes has a directory named /var, but it is unique to each node, not shared.

**Caution**

Users should not store research data in their home directories, which reside on the /home or /usr partition. This is a small partition, and users have a very limited disk quota. You can monitor your disk usage in /home by running `quota -f /home` on the CentOS Linux clusters, Avi and Mortimer:

```
Linux login.mortimer bacon ~ 402: quota -f /home
Disk quotas for user bacon (uid 504):
           Filesystem blocks  quota  limit  grace files  quota  limit  grace
     /dev/sda7  72332  262144  262144    4178     0      0
```

The FreeBSD cluster, Peregrine, uses the ZFS filesystem. The command for checking quotas here is `zfs get userquota@your-login-name`.

```
FreeBSD login.peregrine bacon ~ 403: zfs get userquota@bacon
NAME      PROPERTY     VALUE     SOURCE
zroot     userquota@bacon none local
zroot/ROOT userquota@bacon none local
zroot/ROOT/default userquota@bacon none local
zroot/pxe-images userquota@bacon none local
zroot/pxeserver userquota@bacon none local
zroot/share1 userquota@bacon 227 local
zroot/tmp   userquota@bacon 256M local
zroot/usr   userquota@bacon none local
zroot/usr/home userquota@bacon 256M local
zroot/usr/ports userquota@bacon none local
zroot/usr/src userquota@bacon none local
zroot/var userquota@bacon none local
zroot/var/cache userquota@bacon none local
zroot/var/crash userquota@bacon none local
zroot/var/log userquota@bacon none local
zroot/var/mail userquota@bacon 256M local
zroot/var/tmp userquota@bacon 256M local
```

All research data should be stored on one of the large RAIDs (Redundant Arrays of Independent Disks), which provide much greater storage capacity.

A directory is provided on one of the RAID partitions for each new user. A symbolic link to this directory is created in your home directory as ~/Data. Hence, you can cd into this directory by typing `cd ~/Data`

You can see where the link points by typing `ls ~`.

### 6.5 Policies and Guidelines

- The development servers utilize ePanther login credentials, i.e. you can use the same login and password as you go for UWM email and other services.
Because of the security risk of ePanther credentials, the critical nature of the clusters, and the higher risk posed by allowing shell log-ins, the clusters to not utilize ePanther credentials and instead have their own local passwords with high quality standards.

- Passwords must meet quality standards and must be changed periodically. You will be notified by the system when your password expires.
  For convenience, password changes can be performed for the entire cluster by using the `cluster-passwd` command on the head node, rather than using the standard `passwd` command on each node.
- All jobs (serial and parallel) run on the cluster must be submitted through the scheduler.
- Program compilation should also be scheduled, to ensure that the program is built in the same environment in which it will run. Cluster compute nodes have a different configuration than the head node. Programs compiled on one may not work correctly on the other.
- All data used for computations should be stored on one of the file servers. For your convenience, a link to your directory in one of the shared directories is automatically added to each user’s home directory as ~/Data.
- Each user has a small disk quota for their home directory. and a large quota on the file servers.
  You can view your quotas by running `quota -v`.
- Jobs default (soft) limits and maximum (hard) limits on memory and other resources. Default limits are used when the user fails to specify the resource requirements for a job. Default limits are set low, to prevent jobs with unknown resource requirements from reserving more resources than they require.
  Default limits for Peregrine and Avi are shown in Table 6.2 and Table 6.3.
  Default limits can be overridden by specifying resource requirements in the job submission script. Users can set resource limits lower or higher than the default limits, up the maximum limits shown in Table 6.2.
  Users are encouraged to set limits as low as possible for all jobs, so that the scheduler can use available resources for other jobs. Actual use + 10% is generally a good rule of thumb for memory requirements.
  Users should only exceed the default limit on cores when the load on the cluster is low, and submit any jobs beyond the limit to one of the “nice” partitions, where they can be preempted by higher priority jobs.

<table>
<thead>
<tr>
<th>Resource</th>
<th>Default Limit</th>
<th>Maximum Limit</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cores</td>
<td>72</td>
<td>72</td>
</tr>
<tr>
<td>CPU time (total for entire job)</td>
<td>Unlimited</td>
<td>Unlimited</td>
</tr>
<tr>
<td>Memory (per process)</td>
<td>256 MB</td>
<td>All available</td>
</tr>
<tr>
<td>Wall time</td>
<td>Unlimited</td>
<td>Unlimited</td>
</tr>
</tbody>
</table>

Table 6.2: Job Limits

<table>
<thead>
<tr>
<th>Resource</th>
<th>Default Limit</th>
<th>Maximum Limit</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cores</td>
<td>200</td>
<td>500</td>
</tr>
<tr>
<td>CPU time (total for entire job)</td>
<td>Unlimited</td>
<td>Unlimited</td>
</tr>
<tr>
<td>Memory (per process)</td>
<td>256 MB</td>
<td>All available</td>
</tr>
<tr>
<td>Wall time</td>
<td>Unlimited</td>
<td>Unlimited</td>
</tr>
</tbody>
</table>

Table 6.3: Job Limits

### 6.6 Using Available Software

Most open source software is installed via FreeBSD ports on our FreeBSD systems and pkgsrc on our CentOS systems.

You can use these systems to install the exact same software on your own computers. Virtual machine images for FreeBSD and CentOS are available here for those who want an easy way to utilize these systems on their own computers.

To list available software installed via these package managers, do one of the following:
1. If you are on a cluster (Avi, Mortimer, or Peregrine), run `slurm-shell` to log you into a compute node, where scientific software is accessible. This step is not necessary (or even possible) on development/testing servers such as sciprog and sciprog2.

2. On FreeBSD systems (sciprog and peregrine), run `pkg info`. As with any other Unix command, you can pipe the output through filters if desired.

   FreeBSD login.peregrine bacon ~ 401: slurm-shell
   FreeBSD compute-001.peregrine bacon ~ 401: pkg info | fgrep bowtie
   bowtie-1.1.2 Ultrafast, memory-efficient short read aligner
   bowtie2-2.2.8 Ultrafast, memory-efficient short read aligner

On CentOS systems, first load the module for the pkgsrc collection of interest. Then run `pkgin`.

   Linux login.avi bacon ~ 135: slurm-shell
   Linux compute-1-12.avi bacon ~ 135: module load pkgsrc/2016Q1
   Linux compute-1-12.avi bacon ~ 136: pkgin list | fgrep bowtie
   bowtie2-2.2.7 Ultrafast, memory-efficient short read aligner

Commercial software and a few open source applications are installed under `/sharedapps`. To see what is available there, simply browse using `ls`.

   Linux login.avi bacon ~ 136: ls /sharedapps/
   Admin/ Commercial-Software/ lost+found/ pkg-2016Q1/ pkgsrc-3/
   Backup/ Dist/ LS/ pkg-3/ readme.txt*
   Base-GCC/ Examples/ modulefiles/ pkgsrc-1/ SFS/
   cache/ ICC/ pkg-1/ pkgsrc-2/ SPH/
   CEAS/ Logs/ pkg-2/ pkgsrc-2016Q1/
   Linux login.avi bacon ~ 137: ls /sharedapps/CEAS/
   Abaqus/ ANSYS-15/ avdeev/ Dist/ get-uid* helwany/ modulefiles@
   amano/ ANSYS-17.1/ cyuan/ get-gid* hanson/ kimcs/ woehl/

### 6.7 Code Development and Testing

Production clusters are not a good place to develop and test new code.

- Code development and testing is much easier in a simpler environment such as a workstation, laptop, or development server.
- You do not need a cluster to test parallel code. It can be fully tested and debugged on a single machine, even using a single core, using small test cases.
- Running buggy code on a production cluster may cause problems for other users. Test runs compete for resources that needed for actual research computation and program bugs can have a serious impact on compute nodes, such as slowing them down or consuming all available memory.

Programs should not be run on the production clusters unless they are expected to succeed.

New code should be developed and fully tested on your own hardware or using the development and testing servers managed by UWM Research Computing Support.

You will want to develop small test cases that complete in a few minutes, while testing all the same program features that you will use in real runs. This will allow you to progress through the learning and testing stages quickly.

The development and testing servers are also used for evaluating, learning and testing new software installed by Research Computing Support, before the same software is installed on the clusters. This speeds up the evaluation, learning, and testing processes and avoids impacting other cluster users with program bugs and user errors.
6.8 Off-campus Access

Research computing resources are only accessible from within the campus network. To connect from home or other locations off campus, you will need to use the campus VPN (Virtual Private Network), a tool that makes it appear that your computer is on campus.

UWM uses the Juniper networks VPN server. It requires installing a client program on your computer and then using it to connect to the VPN server on campus. Instructions are provided at https://kb.uwm.edu/uwmhd/page.php?id=57777.

Note The Research Computing support group does not directly support the VPN service. If you experience any issues with the VPN service, please contact the help desk at http://www4.uwm.edu/technology/help/campus/help-request-form.cfm

There is also an open source client program that works with the Juniper VPN, called "OpenConnect".

Caution There is no official support for OpenConnect at UWM. The instructions below are provided as-is for your benefit and we can offer only limited advice regarding issues with OpenConnect.

The OpenConnect client runs on most Unix-compatible systems and can be installed via most package managers, such as Debian packages, FreeBSD ports, MacPorts, etc.

Note OpenConnect must be built with GNUTLS support in order to function with UWM's VPN server. This is the default in some package managers, but not in others.

For RHEL/CentOS Linux and Mac users, we recommend using the binary pkgsrc collection from Joyent Cloud Services. Instructions for installing the pkgsrc system are available at http://uwm.edu/hpc/software-management/ under "Binary Packages for Research Computing". After installing the pkgsrc system, simply type the following to install openconnect:

    pkgin install openconnect

FreeBSD users should use the ports system to install from source with TLS support.

    shell-prompt: cd /usr/ports/security/openconnect
    shell-prompt: make config
After installing OpenConnect, run the following as the root user on your computer:

```shell
shell> openconnect --juniper vpn.uwm.edu
```

Below is an example session:

```
ROOT@imacbsd.acadix ~bacon 129 # openconnect --juniper vpn.uwm.edu
WARNING: Juniper Network Connect support is experimental. It will probably be superseded by Junos Pulse support.
GET https://vpn.uwm.edu/
Connected to 129.89.207.228:443
SSL negotiation with vpn.uwm.edu
Server certificate verify failed: signer not found
Certificate from VPN server "vpn.uwm.edu" failed verification.
Reason: signer not found
Enter 'yes' to accept, 'no' to abort; anything else to view: yes
Connected to HTTPS on vpn.uwm.edu
Got HTTP response: HTTP/1.1 302 Found
GET https://vpn.uwm.edu/dana-na/auth/url_default/welcome.cgi
SSL negotiation with vpn.uwm.edu
Server certificate verify failed: signer not found
Connected to HTTPS on vpn.uwm.edu
frmLogin
username:bacon
password:
POST https://vpn.uwm.edu/dana-na/auth/url_default/login.cgi
SSL negotiation with vpn.uwm.edu
Server certificate verify failed: signer not found
Connected to HTTPS on vpn.uwm.edu
Got HTTP response: HTTP/1.1 302 Moved
```
GET https://vpn.uwm.edu/dana-na/auth/url_default/welcome.cgi?p=user-confirm&id=←
SSL negotiation with vpn.uwm.edu
Server certificate verify failed: signer not found
Connected to HTTPS on vpn.uwm.edu
POST https://vpn.uwm.edu/dana-na/auth/url_default/login.cgi
SSL negotiation with vpn.uwm.edu
Server certificate verify failed: signer not found
Connected to HTTPS on vpn.uwm.edu
Got HTTP response: HTTP/1.1 302 Moved
GET https://vpn.uwm.edu/dana/home/index.cgi
SSL negotiation with vpn.uwm.edu
Server certificate verify failed: signer not found
Connected to HTTPS on vpn.uwm.edu
SSL negotiation with vpn.uwm.edu
Server certificate verify failed: signer not found
Connected to HTTPS on vpn.uwm.edu
SSL negotiation with vpn.uwm.edu
Server certificate verify failed: signer not found
Connected to HTTPS on vpn.uwm.edu
Connected as 10.254.254.62, using SSL
add host 129.89.207.228: gateway 192.168.0.1
route: writing to routing socket: No such process
delete net 10.254.254.62: gateway 10.254.254.62 fib 0: not in table
route: writing to routing socket: No such process
delete net 172.20.0.0: gateway 10.254.254.62 fib 0: not in table
add net 172.20.0.0: gateway 10.254.254.62
route: writing to routing socket: No such process
delete net 172.18.0.0: gateway 10.254.254.62 fib 0: not in table
add net 172.18.0.0: gateway 10.254.254.62
route: writing to routing socket: No such process
delete net 129.89.0.0: gateway 10.254.254.62 fib 0: not in table
add net 129.89.0.0: gateway 10.254.254.62
route: writing to routing socket: No such process
delete net 10.0.0.0: gateway 10.254.254.62 fib 0: not in table
add net 10.0.0.0: gateway 10.254.254.62
route: writing to routing socket: No such process
delete net 129.89.10.1: gateway 10.254.254.62 fib 0: not in table
add net 129.89.10.1: gateway 10.254.254.62
route: writing to routing socket: No such process
delete net 129.89.10.2: gateway 10.254.254.62 fib 0: not in table
add net 129.89.10.2: gateway 10.254.254.62
route: writing to routing socket: No such process
delete host 129.89.207.228: gateway 192.168.0.1
ifconfig: ioctl (SIOCAIFADDR): Destination address required
User canceled (SIGINT); exiting.

At this point you are connected to the VPN. Leave OpenConnect running in this terminal window while you use the VPN. When you are done, simply press Ctrl+c to terminate the connection, as follows:

^Cdelete net 172.20.0.0: gateway 10.254.254.62
delete net 172.18.0.0: gateway 10.254.254.62
delete net 129.89.0.0: gateway 10.254.254.62
delete net 10.0.0.0: gateway 10.254.254.62
delete net 129.89.10.1: gateway 10.254.254.62
delete net 129.89.10.2: gateway 10.254.254.62
delete host 129.89.207.228: gateway 192.168.0.1
ifconfig: ioctl (SIOCAIFADDR): Destination address required
User canceled (SIGINT); exiting.
Chapter 7

Introduction to Unix

Before You Begin
If you think the word "Unix" refers to Sumerian servants specially "trained" to guard a harem, you've come to the right place. This chapter is designed as a tutorial for users with little or no Unix experience. If you are following this guide as part of an ungraded workshop, please feel free to work together on the exercises in this text. It would be very helpful if experienced users could assist less experienced users during the "practice breaks" in order to keep the class moving forward and avoid leaving anyone behind.

7.1 Aw, Man... I have to Learn a Another Operating System?

Well, yeah, but it's the last time, I promise. As you'll see in the sections that follow, once you've learned to use Unix, you'll be able to use your new skills on virtually any computer. Over time you'll get better and better at it, and never have to start over from scratch again. Read on to find out why.

Simplicity is the ultimate sophistication.
- Leonardo da Vinci

Most people make most things far more complicated than they need to be. Especially so with engineers and computer scientists. The original design of Unix is an exception to this rule. The developers were wise and tried to make everything as simple and elegant as possible.

Unix is not hard to learn. You may have gotten the impression that it's a complicated system meant for geniuses while listening to geniuses talk about it. Don't let them fool you, though. The genius ego compels every genius to make things sound really hard, so you'll think they're smarter than you.

Unix has grown immensely since it was created, but the reality is, you don't need to know a lot in order to use Unix effectively. The average Unix user can learn almost everything they'll need to know in a day or two. You can become more sophisticated over time if you want, but most Unix users don't really need to. It may be better to stick to the K.I.S.S. (Keep It Simple, Stupid) philosophy and focus on becoming resourceful using the basic tools, rather than accumulating a lot of knowledge that you'll rarely use.

Unix is designed to be as simple as possible and to allow you to work as fast as possible. Many other systems will slow you down by requiring you to use cumbersome user interfaces or spend time learning new proprietary methods. As you become a master of Unix, your productivity will be limited only by the speed of the hardware and programs you run.
7.2 What is Unix?

7.2.1 It’s All the Same...

Unix began as the trade name of an operating system developed at AT&T Bell Labs around 1970. It quickly became the model on which most subsequent operating systems have been based. Eventually, "Unix" came into common use to refer to any operating system mimicking the original Unix, much like Band-Aid is now used to refer to any adhesive bandage purchased in a drug store.

Over time, formal standards were developed to promote compatibility between the various Unix-like operating systems, and eventually, Unix ceased to be a trade name. Today, the name Unix officially refers to a set of standards to which most operating systems conform.

Look around the room and you will see many standards that make our lives easier. (Wall outlets, keyboards, USB ports, light bulb sockets, etc.) All of these standards make it possible to buy interchangeable devices from competing companies. This competition forces the companies to offer better value. They need to offer a lower price and/or better quality than their competition in order to stay in business.

In a nutshell, Unix is every operating system you’re likely to use except Microsoft Windows. Table 7.1 provides links to many Unix-compatible operating systems. This is not a comprehensive list. Many more Unix-like systems can be found by searching the web.

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>URL</th>
</tr>
</thead>
<tbody>
<tr>
<td>CentOS GNU/Linux</td>
<td>Free</td>
<td><a href="https://en.wikipedia.org/wiki/CentOS">https://en.wikipedia.org/wiki/CentOS</a></td>
</tr>
<tr>
<td>Debian GNU/Linux/FreeBSD</td>
<td>Free</td>
<td><a href="https://en.wikipedia.org/wiki/Debian">https://en.wikipedia.org/wiki/Debian</a></td>
</tr>
<tr>
<td>FreeBSD</td>
<td>Free</td>
<td><a href="https://en.wikipedia.org/wiki/FreeBSD">https://en.wikipedia.org/wiki/FreeBSD</a></td>
</tr>
<tr>
<td>GhostBSD</td>
<td>Free</td>
<td><a href="https://en.wikipedia.org/wiki/GhostBSD">https://en.wikipedia.org/wiki/GhostBSD</a></td>
</tr>
<tr>
<td>illumos</td>
<td>Free</td>
<td><a href="https://en.wikipedia.org/wiki/Illumos">https://en.wikipedia.org/wiki/Illumos</a></td>
</tr>
<tr>
<td>JunOS (Juniper Networks)</td>
<td>Commercial</td>
<td><a href="https://en.wikipedia.org/wiki/Junos">https://en.wikipedia.org/wiki/Junos</a></td>
</tr>
<tr>
<td>Linux Mint</td>
<td>Free</td>
<td><a href="https://en.wikipedia.org/wiki/Linux_Mint">https://en.wikipedia.org/wiki/Linux_Mint</a></td>
</tr>
<tr>
<td>MirOS BSD</td>
<td>Free</td>
<td><a href="https://en.wikipedia.org/wiki/MirOS_BSD">https://en.wikipedia.org/wiki/MirOS_BSD</a></td>
</tr>
<tr>
<td>NetBSD</td>
<td>Free</td>
<td><a href="https://en.wikipedia.org/wiki/NetBSD">https://en.wikipedia.org/wiki/NetBSD</a></td>
</tr>
<tr>
<td>OpenBSD</td>
<td>Free</td>
<td><a href="https://en.wikipedia.org/wiki/OpenBSD">https://en.wikipedia.org/wiki/OpenBSD</a></td>
</tr>
<tr>
<td>OpenIndiana</td>
<td>Free</td>
<td><a href="https://en.wikipedia.org/wiki/OpenIndiana">https://en.wikipedia.org/wiki/OpenIndiana</a></td>
</tr>
<tr>
<td>TrueOS</td>
<td>Free</td>
<td><a href="https://en.wikipedia.org/wiki/TrueOS">https://en.wikipedia.org/wiki/TrueOS</a></td>
</tr>
<tr>
<td>Redhat Enterprise Linux</td>
<td>Commercial</td>
<td><a href="https://en.wikipedia.org/wiki/Red_Hat_Enterprise_Linux">https://en.wikipedia.org/wiki/Red_Hat_Enterprise_Linux</a></td>
</tr>
<tr>
<td>SCO UnixWare</td>
<td>Commercial</td>
<td><a href="https://en.wikipedia.org/wiki/UnixWare">https://en.wikipedia.org/wiki/UnixWare</a></td>
</tr>
<tr>
<td>SmartOS</td>
<td>Free</td>
<td><a href="https://en.wikipedia.org/wiki/SmartOS">https://en.wikipedia.org/wiki/SmartOS</a></td>
</tr>
<tr>
<td>Solaris</td>
<td>Commercial</td>
<td><a href="https://en.wikipedia.org/wiki/Solaris_(operating_system)">https://en.wikipedia.org/wiki/Solaris_(operating_system)</a></td>
</tr>
<tr>
<td>SUSE Enterprise Linux</td>
<td>Commercial</td>
<td><a href="https://en.wikipedia.org/wiki/SUSE_Linux_Enterprise_Desktop">https://en.wikipedia.org/wiki/SUSE_Linux_Enterprise_Desktop</a></td>
</tr>
<tr>
<td>Ubuntu Linux (See also</td>
<td>Free</td>
<td><a href="https://en.wikipedia.org/wiki/Ubuntu_(operating_system)">https://en.wikipedia.org/wiki/Ubuntu_(operating_system)</a></td>
</tr>
<tr>
<td>Kubuntu, Lubuntu, Xubuntu)</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 7.1: Partial List of Unix Operating Systems

Note: Apple’s Mac OS X has many proprietary extensions, including Apple’s own user interface, but is almost fully Unix-compatible and can be used much like any other Unix system by simply choosing not to use the Apple extensions.

In addition to the operating systems above, most smart phones run Unix-based operating systems. Apple’s iOS is largely based on BSD and Android is based on Linux.
Unix is historically connected with other standards such as X/Open XPG and POSIX (Portable Operating System Interface based on Unix). The Unix-related standards are fact the only open standards in existence for operating systems. For the official definition of Unix and associated standards, see the Open Group website: http://www.unix.org/what_is_unix.html.

The Unix standards serve the same purpose as all standards; to foster collaboration, give the consumer freedom of choice, reduce unnecessary learning time, and annoy developers who would rather ignore what everyone else is doing and reinvent the wheel on their employer’s dime.

Unix standards make things interchangeable in the same way as many other standards, such as power plugs, USB, cell phone SIM cards, etc.

*Note* When you develop programs for any Unix-compatible operating system, those programs can be easily used by people running any other Unix-compatible system. Most Unix programs can even be used on a Microsoft Windows system with the aid of a compatibility layer such as Cygwin (See Section 7.5.1).

*Note* Once you’ve learned to use one Unix system, you’re ready to use any of them. Hence, Unix is the last system you’ll ever need to learn!

Unix systems run on everything from your cell phone to the world’s largest supercomputers. Unix is the basis for Apple’s iOS, the Android mobile OS, embedded systems such as networking equipment and robotics controllers, most PC operating systems, and many large mainframe systems.

Many Unix systems are completely free (as in free beer) and can run thousands of high quality free software packages.

It’s a good idea to regularly use more than one Unix system. This will make you aware of what they all have in common and what the differences (extensions of each) are.

7.2.2 Operating System, or Religion?

The whole point of the Unix standard, like any other standard, is freedom of choice.

However, you won’t have any trouble finding evangelists for a particular brand of Unix-compatible operating system on whom this point is lost. "Discussions" about the merits of various Unix implementations often involve emotional outbursts and a fair amount of cussing.

If you step back and ask yourself what kind of person gets emotionally attached to a piece of software, you’ll realize who you should and should not be taking advice from. Rational people will keep an open mind and calmly discuss the objective measures of an OS, such as performance, reliability, security, easy of maintenance, specific capabilities, etc. They will also back up their statements with facts.

If someone tells you that a particular operating system "isn’t worth using", "is way behind the times", or "sucks wads", rather than asking you what you need and objectively considering the alternatives, you can safely ignore them.

Evangelists are typically pretty easy to spot. They will magically assess your needs without asking you a single question and proceed to explain aggressively why you should be using their favorite operating system.

Ultimately, the system that most easily runs your programs to your satisfaction is the best one for you. That could turn out to be Cygwin, FreeBSD, Linux, Mac OS X, or any other.

Someone who knows what they’re doing and truly wants to help you will always begin by asking you questions in order to better understand your needs. They will be knowledgeable about multiple alternatives and inform you about the capabilities of each one that seems to match your needs.

7.2.3 The Unix Standard API

Programmers cost money. This is a problem in the computer industry for which we haven’t found a solution. Even if we keep them locked in a basement for days at a time and only pay them for half the hours they work (which many of them oddly find
perfectly acceptable), they still need to be fed and watered occasionally, and paid a reasonably competitive salary so they won’t leave and go work in someone else’s basement.

Unix systems adhere to an *application program interface* (API) standard, which means that programs written for one Unix-based system can be run on any other with little or no modification, even if it runs on completely different hardware. For example, programs written for an Intel/AMD-based Linux system will also run a PowerPC based Mac, a Solaris system running on a Sparc processor, or FreeBSD on an ARM processor.

An API defines a set of functions (subprograms) used to request services from the operating system, such as opening a file, allocating memory, running another program, etc.

These functions are the same on all Unix systems, but some of them are different on Windows and other non-standard systems. For example, to open a file in a C program on any Unix system, one could use the fopen() function:

```c
FILE *fopen(const char *filename, const char *mode);
```

Microsoft compilers also support fopen(), but also provide another function for the same purpose that won’t work on other systems:

```c
errno_t fopen_s(FILE** pFile, const char *filename, const char *mode);
```

**Note** Microsoft claims that fopen_s() is more secure, which is debatable. Note however, that even if this is true, the existing fopen() function itself could have been made more secure rather than creating a separate, non-standard function that does the same thing.

Here are a few other standard Unix functions that can be used in programs written in C and most other compiled languages. These functions can be used on any Unix system, regardless of the type of hardware running it. Some of these may also work in Windows, but for others, Windows uses a completely different function to achieve the same goal.

```c
chdir()               // Change current working directory
exec()                // Load and run another program
mkdir()               // Create a directory
unlink()              // Remove a file
sleep()               // Pause execution of the process
DisplayWidth()        // Get the width of the screen
```

Because the Unix API is platform-independently, it is also possible to compile and run most Unix programs on Windows with the aid of a *compatibility layer*, software that bridges the difference between two platforms. ( See Section 7.5.1 for details. ) It is not generally possible to compile and run Windows software on Unix, however, since Windows has many PC-specific features.

Since programs written for Unix can be run on almost any computer, including Windows computers, they will probably never have to be rewritten in order to run somewhere else!

Programs written for non-Unix platforms will only run on that platform, and will have to be rewritten (at least partially) in order to run on any other system. This leads to an enormous waste of man-hours that could have gone into creating something new.

They may also become obsolete as they proprietary systems for which they were written evolve. For example, most programs written for MS DOS and Windows 3.x are no longer in use today, while programs written for Unix around that same time will still work on modern Unix systems.

Of course, if you had a lot of fun writing a program the first time, you may want to do it again. In that case, you won’t want to use Unix, since it would take away at least half your fun.

7.2.4 Shake Out the Bugs

Another advantage of programming on standardized platforms is the ability to easily do more thorough testing.

Compiling and running a program on multiple operating systems and with multiple compilers will almost always expose bugs that you were unaware of while running it on the original development system. The same bug will have different effects on
different operating systems, with different compilers or interpreters, or with different compile options (e.g. with and without optimization).

For example, an errant array subscript or pointer might cause corruption in a non-critical memory location in some environments, while causing the program to crash in others.

A program may seem to be fine when you compile it with Clang and run it on your Mac, but may not compile, or may crash when compiled with GCC on a Linux machine.

Finding bugs now may save you from the stressful situation of tracking them down under time pressure later, with an approaching grant deadline. A bug that was invisible on your Mac for the test cases you’ve used could also show up on your Mac later, when you run the program with different inputs.

Developing for the Unix API makes it easy to test on various operating systems and with different compilers. There are many free BSD and Linux based systems, as well as free compilers such as Clang and GCC. Most of them can be run in a virtual machine (Section 7.22), so you don’t even need to have another computer for the sake of program testing.

Take advantage of this easy opportunity to stay ahead of program bugs, so they don’t lead to missed deadlines down the road.

7.2.5 The Unix Standard UI

Let’s face it: Most people don’t like to learn new things. At least not about computers. Unix can help with this, too.

All Unix systems support the same basic set of commands, which conform to standards so that they behave the same way everywhere. So, if you learn to use FreeBSD, most of that knowledge will directly apply to Linux, Mac OS X, Solaris, etc.

Another part of the original Unix design philosophy was to do everything in the simplest way possible. As you learn Unix, you will likely find some of its features befuddling at first. However, upon closer examination, you will often come to appreciate the elegance of the Unix solution to a difficult problem. If you’re observant enough, you’ll learn to apply this Zen-like simplicity to your own work, and maybe even your everyday life. Who knows, mastering Unix could even help you attain enlightenment someday.

You will also gradually recognize a great deal of consistency between various Unix commands and functions. For example, many Unix commands support a \(-v\) (verbose) flag to indicate more verbose output, as well as a \(-q\) (quiet) flag to indicate no unnecessary output. Over time, you will develop an intuitive feel for Unix commands, become adept at correctly guessing how things work, and feel downright God-like at times.

Unix documentation also follows a few standard formats, which users quickly get used to, making it easier to learn new things about commands on any Unix system.

The consistency provided by Unix standards minimizes the amount of knowledge Unix users need in order to effectively utilize the numerous Unix systems available.

In a nutshell, the time and effort you spend learning any Unix system will make it easy to use any other in the future. You need only learn Unix once, and you’ll be proficient with many different implementations such as FreeBSD, Linux, and Mac OS X.

7.2.6 Freedom of Choice

Unix standards are designed to give the user as much freedom of choice as possible. Unix users can run their programs on virtually any type or brand of hardware, and switch at will when a better or cheaper option appears.

As a case in point, until the early 1990’s, most Unix systems were high-end workstations or minicomputers costing $10,000 or more. Many of the same programs that ran on those systems are now running on commodity PCs and even laptops that cost a few hundred dollars. In fact, at this very moment I’m editing this chapter on an old ThinkPad someone else threw away, which is now running FreeBSD with the XFCE desktop environment.

Another of the main design goals of Unix is to stay out of the user’s way. With freedom comes responsibility, though. A common quip about Unix is that it gives us the freedom to shoot ourselves in the foot. Unix does a lot to protect users from each other, but very little to protect users from themselves. This usually leads to some mistakes by new users, but most users quickly become conditioned to be more careful and come to prefer the freedom Unix offers over more restrictive, cumbersome systems.
7.2.7 Fast, Stable and Secure

Since Unix systems compete directly with each other to win and retain users running the same programs, developers are highly motivated to optimize objective measures of the system such as performance, stability, and security.

Most Unix systems operate near the maximum speed of the hardware on which they run. Unix systems typically respond faster than other systems on the same hardware and run intensive programs in less time. Many Unix systems require far fewer resources than non-Unix systems, leaving more disk and memory for use by your programs.

Unix systems may run for months or even years without being rebooted. Software installations almost never require a reboot, and even most security updates can be applied without rebooting. As a professional systems manager, I run the risk of forgetting that some of my Unix systems exist because I haven’t touched them for so long. I’m occasionally reminded when a sleep-deprived graduate student nods off at his desk and knocks the monitor to the floor with a nervous twitch.

Stability is critical for research computing, where computational models often run for weeks or months. Users of non-Unix operating systems often have to choose between killing a process that has been running for weeks and neglecting critical security updates that require a reboot.

Very few viruses or other malware programs exist for Unix systems. This is in part due to the inherently better security of Unix systems and in part due to a strong tradition in the Unix community of discouraging users from engaging in risky practices such as running programs under an administrator account and installing software from pop-ups on the web.

7.2.8 Sharing Resources

Your mom probably told you that it’s nice to share, but did you know it’s also more efficient?

One of the major problems for researchers in computational science is managing their own computers. Most researchers aren’t very good at installing operating systems, managing software, apply security updates, etc., nor do they want to be. Unfortunately, they often have to do these things in order to conduct computational research. Computers managed by a tag-team of researchers usually end up full of junk software, out-of-date, full of security issues, and infected with malware.

Some universities have staff to help with research computing support, but most do not. Good IT guys are expensive and hard to find, so it takes a critical mass of demand from researchers to motivate the creation of a research computing support group.

Since Unix is designed from the ground up to be accessed remotely, Unix creates an opportunity to serve researchers’ needs far more cost-effectively than individual computers for each researcher. A single Unix machine on a modern PC can support dozens or even hundreds of users at the same time.

IT staff managing one or a few central Unix hosts can serve the needs of many researchers, freeing the researchers to focus on what they do best. All the researchers need is a computer that can connect to the central Unix host, and the systems managers of the host can take care of all the hard stuff.

7.3 Self-test

1. Is Unix an operating system? Why or why not?
2. Which mainstream operating systems are Unix compatible and which are not?
3. What is the advantage of the Unix API over the APIs of non-Unix operating systems?
4. What is the advantage of the Unix UIs over the UIs of non-Unix operating systems?
5. What is the major design goal of the Unix standard?
7.4 Unix User Interfaces

A user interface, or UI, refers to the software that allows a person to interact with the computer. The UI provides the look and feel of the system, and determines how easily and efficiently it can be used. (Note that ease of use and efficiency are not the same!)

The term "Windows" refers to a specific proprietary operating system, and implies all of the features of that system including the API and the UI. When people think of Windows, they think of the Start menu, the Control Panel, etc.

Likewise, "Macintosh" refers to a specific product and invokes images of the "Dock" and a menu bar at the top of the screen vs. attached to a window.

The term "Unix", on the other hand, implies an API, but does not imply a specific UI. There are many UIs available for Unix systems. In fact, a computer running Unix can have many UIs installed, and each user can choose the one they want when they log in.

7.4.1 Graphical User Interfaces (GUIs)

Unlike Microsoft Windows, which has a unique look and feel, there are many different GUIs (pronounced goo-ey) available for Unix. Some of the more popular ones include KDE, Gnome, XFCE, LXDE, OpenBox, CDE, and Java Desktop.

![A FreeBSD system running Gnome desktop.](image-url)
A FreeBSD system running KDE desktop.

A FreeBSD system running Lumina desktop.
A FreeBSD system running XFCE desktop.

Practice Break
If you have access to a Unix GUI, log into your Unix system via the GUI interface now.

All Unix GUIs are built on top of the X11 networked graphics API. As a result, all Unix systems have the inherent ability to display graphics on other Unix systems. I.e., you can remotely log into another Unix computer over a network and run graphical programs that display output wherever you’re sitting.

Note This is not the same as a remote desktop system, which only mirrors the console display on a remote system. Unix systems allow multiple users in different locations to run graphical programs independent of each other. In other words, Unix supports multiple independent graphical displays on remote computers.

Most Unix GUIs support multiple virtual desktops, also known as workspaces. Virtual desktops allow a single monitor to support multiple separate desktop images. It’s like having multiple monitors without the expense and clutter. The user can switch between virtual desktops by clicking on a panel of thumbnail images, or in some cases by simply moving the mouse over the edge of the screen.

7.4.2 X11 on Mac OS X

Mac OS X is Unix compatible, derived largely from FreeBSD and the Mach kernel project, with components from GNU and other Unix-based projects.

It differs from more traditional Unix systems like BSD and Linux, though, in that it runs Apple’s proprietary graphical API and GUI. Native OS X programs don’t use the X11 API, but OS X can also run X11-based programs.

Some versions of OS X had the Unix X11 API included, while others need it installed separately. At the time of this writing, X11 on the latest OS X is provided by the XQuartz project, described at https://support.apple.com/en-us/HT201341.
7.4.3 Command Line Interfaces (CLIs): Unix Shells

There are two basic type of user interfaces:

- Menu-driven, where choices are displayed on the screen and the user selects one.
- Command-driven, where the user types commands that they have memorized.

A GUI is a type of menu-driven interface, where the menu items are graphical icons. Some menu systems are simply text. While modern Unix systems have GUIs, much work is still done via the command line, or shell.

Menu-driven systems are much easier to use if you’re new to the system or use it infrequently, but can become cumbersome for everyday use. For example, an ATM (automatic teller machine) with a command-driven interface would likely be unpopular among banking customers.

If a user needs access to dozens or hundreds of features, they cannot all be displayed on the screen as icons at the same time. Hence, it will be necessary to navigate through multiple levels of menus or screens to find the functionality you need. Even if they could be displayed all at once, it would be a nightmare to find the one you want among such clutter.

Because of these limitations, most GUIs support *hot keys*, special key combinations that can be used to access certain features without navigating the menus. Hot keys are often shown in menus alongside the features they activate. For example, Command+q can be used on Mac OS X to terminate most graphical applications.

It is also difficult to automate tasks in a menu-driven system. Some systems have this capability, but most do not, and the method of automating is different for each system.

Perhaps the most important drawback of menu-driven systems is non-existence. Programming a menu system, and especially a GUI, requires a lot of grunt-work and testing. As a result, the vast majority of open source software does not and never will have a GUI interface. Open source developers generally don’t have the time or programming skills to build and maintain a comprehensive GUI interface.

---

**Caution**

If you lack command-line skills, you will be limited to using a small fraction of available open source software. In the tight competition for research grants, those who can use the command-line will usually win.

---

A command line interface, on the other hand, provides instant access to an unlimited number of commands and is easy to automate. We can simply store a sequence of commands in a file, called a script.

A command line interface requires some learning, since we need to memorize some commands in order to use it efficiently. However, we usually need only learn a few commands to get started, and once the basics are learned, a command line interface allows for much greater efficiency and flexibility than a GUI.

The small investment in learning a command line interface can have a huge payoff, and yet many people try to avoid it. The result is usually an enormous amount of wasted effort dealing with limited and poorly designed custom user interfaces before eventually realizing that things would have been much easier had they learned to use the command line in the first place. It’s amazing how much effort people put into avoiding effort...

A *shell* is a program that provides the command line interface. It inputs commands from the user, interprets them, and executes them.

Using a shell, you type a command, press enter, and the command is immediately executed.
Unix Shell

The term shell comes from the view of Unix as three layers of software wrapped around the hardware:

- The innermost layer, which handles all hardware interaction for Unix programs, is called the *kernel*, named after the core of a seed. The Unix kernel effectively hides the hardware from user programs and provides a standard API. This is what allows Unix programs to run on different kinds of computers.
- The middle layer, the libraries, provide a wealth of standard functionality for Unix programmers to utilize. The libraries are like a huge box of Legos that can be used to build all kinds of sophisticated programs.
- The outermost layer, the CLI, is called a shell.

7.4.4 Terminals

All that is needed to use a Unix shell is a keyboard and a screen. In the olden days, these were provided by a simple hardware device called a *terminal*, which connected a keyboard and screen to the system through a simple communication cable. These terminals typically did not have a mouse or any graphics capabilities. They usually had a text-only screen of 80 columns by 24 lines, and offered limited capabilities such as moving the cursor, scrolling the screen, and perhaps a limited number of colors.
Hardware terminals lost popularity with the advent of cheap personal computers, which can perform the function of a terminal, as well as running programs of their own. Terminals have been largely replaced by terminal emulators. A terminal emulator is a simple program that emulates an old style terminal within a window on your desktop.

(A Terminal emulator.)

All Unix systems come with a terminal emulator program. There are also free terminal emulators for Windows, which are discussed in Section 7.6.

For purists who really want to emulate a terminal, there’s Cool Retro Terminal (CRT for short, which also happens to stand for cathode ray tube). This emulator comes complete with screen distortion and jitter to provide a genuine nostalgic 1970s experience.
Once you’re logged in and have a shell running in your terminal window, you’re ready to start entering Unix commands. The shell displays a *prompt* (such as "FreeBSD manta bacon ~ 392:" in the image above) to indicate that it’s waiting for you to enter the next command. The shell prompt can be customized by each user, so it may be different on each Unix system you use.

For clarity, we primarily use the following to indicate a shell prompt in this text:

```
shell-prompt:
```

The actual shell prompt you see may be different on each individual Unix system you use. Including the hostname, user name, and other information in a shell prompt is a common practice and very helpful for users who are logged into multiple Unix systems at the same time from the same desktop. This discourages users from running a command on the wrong computer! Section 7.16 explains how to set your shell prompt in common Unix shells.

The shell prompt in the terminal window above is "FreeBSD manta bacon ~ 392:". At the first prompt, the user entered the command `ls`, which lists the files in the current directory. After `ls` finished, the shell printed another prompt.

To enter a Unix command, you type the command on a single line, edit if necessary (using arrow keys to move around), and press Enter or Return.

Modern Unix shells allow commands to be extensively edited. Assuming your terminal type is properly identified by the Unix system, you can use the left and right arrow keys to move around, backspace and delete to remove characters (Ctrl+h serves as a backspace in some cases), and other key combinations to remove words, the rest of the line, etc. Learning the editing capabilities of your shell will make you a much faster Unix user, so it's a great investment of a small amount of time.
**Practice Break**
Log into your Unix system, open a terminal if necessary, and run the following commands:

```
shell-prompt: ls
shell-prompt: ls /
shell-prompt: ls -al
shell-prompt: mkdir -p Data/IRC
shell-prompt: cd Data/IRC
shell-prompt: nano sample.txt
```

Type in some text, then save the file (press Ctrl+o), and exit nano (press Ctrl+x).

```
shell-prompt: ls
shell-prompt: cat sample.txt
shell-prompt: wc sample.txt
shell-prompt: whoami
shell-prompt: hostname
shell-prompt: uname
shell-prompt: date
shell-prompt: cal
shell-prompt: bc -l
```

```
scale=50
sqrt(2)
8^2
2^8
a=1
b=2
c=1
(-b+sqrt(b^2-4*a*c))/2a
2*a(100000000000)
quit
```

```
shell-prompt: w
shell-prompt: ls /bin
```

### 7.4.6 Self-test

1. What is a UI?
2. What is a GUI?
3. What is the difference between Unix and other operating systems with respect to the GUI?
4. What is a CLI?
5. What are the advantages and disadvantages of a CLI vs. a GUI?
6. What is a shell?
7. What is a kernel?
8. What are libraries?
9. What is a terminal?
11. What is a shell prompt?
7.5 Still Need Windows? Don’t Panic!

For those who need to run software that is only available for Windows, or those who simply haven’t tried anything else yet, there are options for getting to know Unix while still using Windows for your daily work.

Cygwin is a compatibility layer, another layer of software on top of Windows that translates the Unix API to the Windows API. As such, performance is not as good as a native Unix system on the same hardware.

Cygwin may not be ideal for heavy-duty data analysis where optimal performance is required, but it is an excellent system for basic development and testing of Unix code and for interfacing with other Unix systems.

7.5.1 Cygwin: It’s Worth a Shot

Cygwin is a free collection of Unix software, including many system tools from Linux, ported to Windows. It can be installed on any typical Windows machine in a few minutes and allows users to experience a Unix user interface as well as run many popular Unix programs right on the Windows desktop.

To install Cygwin:

1. Download `setup-x86.exe` (32-bit Windows) or `setup-x86_64.exe` (64-bit Windows) from http://www.cygwin.com and save a copy on your desktop or some other convenient location. You will need this program to install additional packages in the future.
2. Run `setup-x86.exe` or `setup-x86_64.exe` and follow the instructions on the screen.
   Unless you know what you're doing, accept the default answers to most questions. Some exceptions are noted below.

3. Unless you know what you're doing, simply choose "Install from Internet".
4. Select where you want to install the Cygwin files and whether to install for all users of this Windows machine.

5. Select where to save downloaded packages. Again, the default location should work for most users.
6. Select a network connection type.

7. Select a download site. It is very important here to select a site near you. Choosing a site far away can cause downloads to be incredibly slow. For users at UWM, the tds.net site, located in Madison, WI, is a good choice.
8. When you reach the package selection screen, select at least the following packages in addition to the basic installation:

- net/openssh
- net/rsync
- x11/xhost
- x11/xinit

This will install the `ssh` command as well as an X11 server, which will allow you to run graphical Unix programs on your Windows desktop. You may not need graphical capabilities immediately, but they will likely come in handy down the road. The rsync package is especially useful if you’ll be transferring large amounts of data back and forth between your Windows machine and remote servers.

Click on the package categories displayed in order to expand them and see the packages under them.
Cygwin can also enable you to do Unix program development on your Windows machine. There are many packages providing Unix development tools such as compilers and editors, as well as libraries. The following is a small sample of common development packages:
Note
Many of these programs are easier to install and update than their counterparts with a standard Windows interface. By running them under Cygwin, you are also practicing use of the Unix interface, which will make things easy for you when need to run them on a cluster or other Unix host that is more powerful than your PC.

- devel/clang (C/C++/ObjC compiler)
- devel/clang-analyzer (Development and debugging tool)
- devel/gcc-core (GNU Compiler Collection C compiler)
- devel/gcc-g++
- devel/gcc-gfortran
- devel/make (GNU make utility)
- editors/emacs (Text editor)
- editors/gvim (Text editor)
- editors/nano (Text editor)
- libs/openmpi (Distributed parallel programming tools)
- math/libopenblas (Basic Linear Algebra System libraries)
- math/lapack (Linear Algebra PACKAGE libraries)
- math/octave (Open source linear algebra system compatible with Matlab(r))
- math/R (Open source statistical language)

9. Most users will want to accept the default action of adding an icon to their desktop and to the Windows Start menu.
When the installation is complete, you will find Cygwin and Cygwin/X folders in your Windows program menu. For a basic Terminal emulator, just run the Cygwin terminal:

If you’d like to change the font size or colors of the Cygwin terminal emulator, just right-click on the title bar of the window:
Within the Cygwin terminal window, you are now running a "bash" Unix shell and can run most common Unix commands such as "ls", "pwd", etc.
If you selected the openssh package during the Cygwin installation, you can now remotely log into other Unix machines, such as the clusters, over the network:

Note If you forgot to select the openssh package, just run the Cygwin setup program again. The packages you select when running it again will be added to your current installation.

If you want to run Unix graphical applications, either on your Windows machine or on a remote Unix system, run the Cygwin/X application:
Note Doing graphics over a network may require a fast connection. If you are logging in from home or over a wireless connection, you may experience very sluggish rendering of windows from the remote host.

Depending on your Cygwin setup, this might automatically open a terminal emulator called "xterm", which is essentially the same as the standard Cygwin terminal, although it has a different appearance. You can use it to run all the same commands you would in the standard Cygwin terminal, including ssh. You may need to use the -X or -Y flag with ssh to enable some remote graphical programs.

Unlike Cygwin Terminal, the xterm supplied with Cygwin/X is preconfigured to support graphical applications. See Section 7.21.2 for details.
Caution Use of the -X and -Y flags could compromise the security of your Windows system by allowing malicious programs on the remote host to display phony windows on your PC. Use them only when logging into a trusted host.

Once you are logged into the remote host from the Cygwin/X xterm, you should be able to run graphical Unix programs.
You can also run graphical applications from the standard Cygwin terminal if you update your start up script. If you are using bash (the Cygwin default shell), add the following line to your .bashrc file:

```
export DISPLAY=unix:0.0
```

You will need to run `source .bashrc` or restart your bash shell after making this change.

If you are using T-shell, the line should read as follows in your .cshrc or .tcshrc:

```
setenv DISPLAY unix:0.0
```

Again, Cygwin is not the ideal way to run Unix programs on or from a Windows machine, but it is a very quick and easy way to gain access to a basic Unix environment and many Unix tools. Subsequent sections provide information about other options besides Cygwin for those with more sophisticated needs.

### 7.5.2 Unix and Windows Together?

You can run Windows and Unix on the same PC simultaneously using a virtual machine, such as Oracle VirtualBox, which is also available for free.

If you decide to make the switch to Unix on your PC, you can continue to run Windows software using a virtual machine, or using the WINE Windows compatibility system, which runs many Windows programs natively on PC-based Unix systems. Virtual machines are described in more detail in Section 7.22.

### 7.6 Logging In Remotely

Virtually all Unix systems allow users to log in and run programs over a network from other locations. This feature is intrinsic to Unix systems, and only disabled on certain proprietary or embedded installations. It is possible to use both GUIs and CLIs in this
fashion, although GUIs may not work well over slow connections such as a typical home Internet service. Different graphical programs have vastly different video speed demands. Some will work fine over a DSL connection, while others will not work well even over the fastest network.

The command line interface, on the other hand, works comfortably on even the slowest network connections.

Logging into a Unix CLI from a remote location is usually done using Secure Shell (SSH).

---

**Caution** Older protocols such as rlogin, rsh, and telnet, should no longer be used due to their lack of security. These protocols transport passwords over the Internet in unencrypted form, so people who manage the gateway computers they pass through can easily read them.

---

### 7.6.1 Unix to Unix

If you want to remotely log in from one Unix system to another, you can simply use the `ssh` command from the command line.

The general syntax of the `ssh` command is:

```
ssh [flags] login-id@hostname
```

**Note** If you plan to run graphical programs on the remote Unix system, you may need to include the `-X` (enable X11 forwarding) or `-Y` (enable trusted X11 forwarding) flag in your `ssh` command.

---

**Caution** Use `-X` or `-Y` only when connecting to trusted computers, i.e. those managed by you or someone you trust. These options allow the remote system to access your display, which can pose a security risk.

---

Examples:

```
shell-prompt: ssh joe@sciprog.ceas.uwm.edu
shell-prompt: ssh joe@login.peregrine.hpc.uwm.edu
```

**Note** For licensing reasons, `ssh` may not be included in basic Linux installations, but it can be very easily added via the package management system of most Linux distributions.

---

Once logged in, you can easily open additional terminals from the command-line if you know the name of the terminal emulator. Simply type the name of the terminal emulator, followed by an `&` to put it in the background. (See Section 7.18.3 for a full explanation of background jobs.) Some common terminal emulators are xterm, rxvt, gnome-terminal, xfce-terminal, konsole, and Terminal.

```
shell-prompt: xfce-terminal &
```

If you are using ssh over a slow connection, such as home DSL/cable, and plan to use X11 programs, it can be very helpful to enable compression, which is enabled by the `-C` flag. Packets are then compressed before being sent over the wire and decompressed on the receiving end. This adds more CPU load on both ends, but reduces the amount of data flowing over the network and may significantly improve the responsiveness of a graphical user interface. Run `man ssh` for details.

```
shell-prompt: ssh -YC joe@login.peregrine.hpc.uwm.edu
```
7.6.2 Windows to Unix

If you’re connecting to a Unix system from a Windows system, you will need to install some additional software.

Cygwin

The Cygwin Unix-compatibility system is free, quick and easy to install, and equips a Windows computer with most common Unix commands, including a Unix-style Terminal emulator. Once Cygwin is installed, you can open a Cygwin terminal on your Windows desktop and use the ssh command as shown above.

The Cygwin installation is very quick and easy and is described in Section 7.5.1.

PuTTY

A more limited method for remotely accessing Unix systems is to install a stand-alone terminal emulator, such as PuTTY, http://www.chiark.greenend.org.uk/~sgtatham/putty/. PuTTY has a built-in ssh client, and a graphical dialog box for connecting to a remote machine. To connect via ssh, simply select the ssh radio button, enter the hostname of the computer you want to connect to, and click the Open button.

7.6.3 Terminal Types

In some cases, you may be asked to specify a terminal type when you log in:

```
TERM=(unknown)
```

Terminal features such as cursor movement and color changes are triggered by sending special codes (characters or character combinations) to the terminal. Pressing keys on the terminal sends codes from the terminal to the computer.

Different types of terminals use different key and screen control codes. PuTTY and most other terminal emulators emulate an "xterm" terminal, so if asked, just type the string "xterm" (without the quotes).

If you fail to set the terminal type, some programs such as text editors will not function. They may garble the screen and fail to recognize special keys such as arrows, page-up, etc. Programs such as ls that simple output a line and then go to the next will generally work fine even if TERM is not set.

You can set the terminal type after logging in, but the methods for doing this vary according to which shell you use, so you may just want to log out and remember to set the terminal type when you log back in.
**Practice Break**

Remotely log into another Unix system using the **ssh** command or PuTTY. Then try starting the **vi** editor:

```
shell-prompt: ls
shell-prompt: ls /
shell-prompt: ls -al
shell-prompt: mkdir -p Data/IRC
shell-prompt: cd Data/IRC
shell-prompt: nano sample.txt
```

Type in some text, then save the file (press Ctrl+o), and exit nano (press Ctrl+x).

```
shell-prompt: ls
shell-prompt: cat sample.txt
shell-prompt: wc sample.txt
shell-prompt: whoami
shell-prompt: hostname
shell-prompt: uname
shell-prompt: date
shell-prompt: cal
shell-prompt: bc -l
```

```
scale=50
sqrt(2)
8^2
2^8
a=1
b=2
c=1
(-b+sqrt(b^2-4*a*c))/2a
2*a(100000000000)
quit
```

```
shell-prompt: w
shell-prompt: ls /bin
```

---

### 7.6.4 Self-test

1. What extra software must you install on Unix systems to allow logging into another Unix system over a network? Explain.
2. What extra software must you install on Unix systems to enable the use of graphical programs over a network? Explain.
3. What is **ssh**? How does it differ from **rsh** and **telnet**?
4. Can Windows users run **ssh**? Explain.
5. Can Windows users run graphical programs on a remote Unix system? Explain.
6. What is the purpose of the **TERM** environment variable? What will happen if it is not set correctly?

---

### 7.7 Unix Command Basics

A Unix command is built from a command name and optionally one or more command line **arguments**. Arguments can be either **flags** or data.

```
ls -a -l /etc /var
```

```
^ ^^^^^ ^^^^^^^^^
| | | | | | | | |
• The *command name* identifies the program to run. For example, the `ls` command names a program that lists the contents of a directory.

• Most commands have optional *flags* (sometimes called *options*) that control the general behavior of the command. By convention, flags begin with a `-` character, just to help the reader distinguish between flags and arguments.

<table>
<thead>
<tr>
<th>Flags</th>
<th>Data Arguments</th>
</tr>
</thead>
</table>

Note Unix systems do not enforce this, but very few commands violate it. Unlike voluntary taxation, voluntary environmental regulations, or voluntary speed limits, the voluntary Unix conventions garner a very high level of conformance. Unix programmers tend to understand the benefits of conventions and don’t have to be forced to follow them.

The flags in the example above have the following meaning:

- `-a` tells `ls` to show "hidden" files (files whose names begin with '.', which `ls` would not normally list).
- `-l` tells `ls` to do a "long listing", which is to show lots of information about each file and directory instead of just the name.

Single-letter flags can usually be combined, e.g. `-a -l` can be abbreviated as `-al`.

Most newer Unix commands also support long flag names, mainly to improve readability of commands used in scripts. For example, in the Unix `zip` command, `-C` and `--preserve-case` are synonymous.

Many commands also accept one or more data arguments, which provide input data to the command, or instruct it where to send output. Such arguments may be the actual input or they may be the names of files or directories that contain input or receive output. The `/etc` and `/var` arguments above are directories to be listed by `ls`. If no data arguments are given to `ls`, it lists the current working directory (described in Section 7.9).

For many Unix commands, the flags must come before the data arguments. A few commands require that certain flags appear in a specific order. Some commands allow flags and data arguments to appear in any order. Unix systems do not enforce any rules regarding arguments. How they behave is entirely up to the creator of the command. However, the vast majority of commands follow conventions, so there is a great deal of consistency in Unix command syntax.

The components of a Unix command are separated by white space (space or tab characters). Hence, if an argument contains any white space, it must be enclosed in quotes (single or double) so that it will not be interpreted as multiple separate arguments.

**Example 7.1 White space in an Argument**

Suppose you have a directory called `My Programs`, and you want to see what’s in it. You might try the following:

```
shell-prompt: ls My Programs
```

The above command fails, because "My" and "Programs" are interpreted as two separate arguments. In fact, the `ls` command will look for two separate directories called "My" and "Programs". In this case, we must use quotes to bind the parts of the directory name together into a single argument. Either single or double quotes are accepted by all common Unix shells. The difference between single and double quotes is covered in Chapter 8.

```
shell-prompt: ls 'My Programs'
shell-prompt: ls "My Programs"
```

As an alternative to using quotes, we can *escape* the space by preceding it with a backslash (`\`) character. This will save one keystroke if there is only one character to be escaped in the text.

```
shell-prompt: ls My\ Programs
```
### Practice Break

Try the following commands:

```
shell-prompt: ls
shell-prompt: ls -al
shell-prompt: ls /etc
shell-prompt: ls -al /etc
shell-prompt: mkdir 'My Programs'
shell-prompt: ls My Programs
shell-prompt: ls "My Programs"
```

---

### 7.7.1 Self-test

1. What are the parts of a Unix command? What separates them?
2. What are command line flags?
3. What are data arguments?
4. What rules do Unix systems enforce regarding the placement of arguments?
5. How do we specify an argument that contains white space?

---

### 7.8 Basic Shell Tools

#### 7.8.1 Common Unix Shells

There are many different shells available for Unix systems. This might sound daunting if you’re new to Unix, but fortunately, like most Unix tools, all the common shells adhere to certain standards.

All of the common shells are derived from one of two early shells:

- Bourne shell (sh) is the de facto basic shell on all Unix systems, and is derived from the original Unix shell developed at AT&T.
- C shell (csh) offers mostly the same features as Bourne shell, but the two differ in the syntax of their scripting languages, which are discussed in Chapter 8. The C shell syntax is designed to be more intuitive and similar to the C language.

Most Unix commands are exactly the same regardless of which shell you are using. Differences will only become apparent when using more advanced command features or writing shell scripts, both of which we will cover later.

Common shells derived from Bourne shell include the following:

- Almquist shell (ash), used as the Bourne shell on many BSD systems.
- Korn shell (ksh), an extended Bourne shell with many added features for user-friendliness.
- Bourne again shell (bash) another extended Bourne shell from the GNU project with many added features for user-friendliness. Used as the Bourne shell on many Linux systems.
- Debian Almquist shell (dash), a reincarnation of ash which is used as the Bourne shell on Debian based Linux systems.

Common shells derived from C shell include the following:

- T shell (tcsh), and extended C shell with many added features for user-friendliness.
- Hamilton C shell, an extended C shell used primarily on Microsoft Windows.

Unix systems differ in which shells are included in the base installation, but most shells can be easily added to any Unix system using the system’s package manager.
7.8.2 Command History

Most shells remember a configurable number of recent commands. This command history is saved to disk, so that you can still recall this session’s commands next time you log in.

The exact mechanisms for recalling those commands varies from shell to shell, but some of the features common to all shells are described below.

Most modern shells support scrolling through recent commands by using the up-arrow and down-arrow keys. Among the popular shells, only very early shells like Bourne shell (sh) and C shell (csh) lack this ability.

Note This feature may not work if your TERM variable is not set properly, since the arrow keys send magic sequences that may differ among terminal types.

The `history` command lists the commands that the shell currently has in memory.

```
shell-prompt: history
```

A command consisting of an exclamation point (!) followed by any character string causes the shell to search for the most recently executed command that began with that string. This is particularly useful when you want to repeat a complicated command.

```
shell-prompt: find Programs -name '*.o' -exec rm -i '{}' 
shell-prompt: !find
```

An exclamation point followed by a number runs the command with that history index:

```
shell-prompt: history
  385 13:42 more output.txt
  386 13:54 ls
  387 13:54 cat /etc/hosts
shell-prompt: !386
```

7.8.3 Auto-completion

In most Unix shells, you need only type enough of a command or argument filename to uniquely identify it. At that point, pressing the TAB key will automatically fill in the rest for you. Try the following:

```
shell-prompt: touch sample.txt
shell-prompt: cat sam<Press the TAB key now>
```

If there are other files in your directory that begin with "sam", you may need to type a few additional characters before the TAB, like ‘p’ and ‘l’ before auto-completion will work.

7.8.4 Command-line Editing

Modern shells allow extensive editing of the command currently being entered. The key bindings for different editing features depend on the shell you are using and the current settings. Some shells offer a selection of different key bindings that correspond to Unix editors such as vi or Emacs.

See the documentation for your shell for full details. Below are some example default key bindings for shells such as bash and tcsh.
7.8.5 Globbing (File Specifications)

There is often a need to specify a large number of files as command line arguments. Typing all of them would be tedious, so Unix shells provide a mechanism called globbing that allows short, simple patterns to match many file names.

These patterns are built from literal text and/or the special symbols called wild cards in Table 7.3.

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Matches</th>
</tr>
</thead>
<tbody>
<tr>
<td>*</td>
<td>Any sequence of characters (including none) except a ‘.’ in the first character of the filename.</td>
</tr>
<tr>
<td>?</td>
<td>Any single character, except a ‘.’ in the first character of the filename.</td>
</tr>
<tr>
<td>[string]</td>
<td>Any character in string</td>
</tr>
<tr>
<td>[c1-c2]</td>
<td>Any character from c1 to c2, inclusive</td>
</tr>
<tr>
<td>(thing1,thing2)</td>
<td>Thing1 or thing2</td>
</tr>
</tbody>
</table>

Table 7.3: Globbing Symbols

Normally, the shell handles these special characters, expanding globbing patterns to a list of matching file names before the command is executed.

If you want an argument containing such special characters to be sent to a command in its raw form, it must be enclosed in single quotes, which are known as hard quotes.

Globbing patterns enclosed in double quotes are treated as if they were not inside quotes! Double quotes are therefore known as soft quotes. Soft quotes do little more than allow white space to be part of an argument. More details on quotes are provided in Section 8.9.

Certain commands, such as find need to receive the pattern as an argument and attempt to do the matching themselves rather than have it done for them by the shell. Therefore, patterns to be used by the find command must be enclosed in hard quotes.

The exemption for a leading ‘.’ prevents accidental matching of hidden files.

```sh
shell-prompt: ls *.txt  # Lists all files ending in *.txt
shell-prompt: ls "*.txt"  # Lists all files ending in *.txt
shell-prompt: ls '.*.txt'  # Fails, unless there is a file called '.*.txt'
shell-prompt: ls ".*.txt"  # Lists hidden files ending in *.txt
shell-prompt: ls [A-Za-z]*  # Lists all files and directories
                          # whose name begins with a letter
shell-prompt: find . -name "*.txt"  # Fails
shell-prompt: find . -name '.*.txt'
shell-prompt: ls *.{c,c++,f90}
```

7.8.6 Self-test

1. Which shells support modern interactive features such as scrolling through previous commands and command-line editing?
2. Show the simplest command to accomplish each of the following:
(a) Show a list of recently executed commands.
(b) Re-execute the most recent command that began with "ls".

3. Show the simplest command to accomplish each of the following:
   (a) Move all the files whose names end with ".c" from the current directory to the directory ./Prog1.
   (b) Remove all the files whose names end with ".o".
   (c) List the contents of all files/directories in the current working directory whose names begin with a `.' followed by a capital letter or lower case letter, and end with a digit.

7.9 Processes

Before You Begin
It is assumed the reader knows what Unix is. If not, please read Section 7.2 before proceeding.

A running program in Unix is known as a process.

Unix is a multitasking system, which means that many programs can be running at any given moment, i.e. there can be many active processes.

When you log in, the system creates a new process to run your shell program.

When you run a program (a command) from the shell, the shell creates a new process to run the program. Hence, you now have two processes running: the shell process and the command’s process.

The process created by the shell to run your command is a child of the shell process.

Naturally, the shell process is called the parent of the command process.

Each process is uniquely identified by an integer serial number called the process ID, or PID.

Unix systems also keep track of each process’s status and resource usage, such as memory, CPU time, etc. Information about your currently running processes can be easily viewed using the ps (process status) command:

<table>
<thead>
<tr>
<th>PID</th>
<th>TTY</th>
<th>TIME</th>
<th>CMD</th>
</tr>
</thead>
<tbody>
<tr>
<td>7147</td>
<td>ttys000</td>
<td>0:00.14</td>
<td>-tcsh</td>
</tr>
<tr>
<td>7438</td>
<td>ttys000</td>
<td>0:01.13</td>
<td>ape notes.dbk unix.dbk</td>
</tr>
<tr>
<td>7736</td>
<td>ttys001</td>
<td>0:00.13</td>
<td>-tcsh</td>
</tr>
</tbody>
</table>

Practice Break
Run the ps command. What processes do you have running?

shell-prompt: ps

Another useful tool is the top command, which monitors all processes in a system and displays system statistics and the top (most active) processes every few seconds. Note that since top is a full-terminal command, it will not function properly unless the TERM environment variable is set correctly.

Practice Break
Run the top command. What processes are using the most CPU time? Type ‘q’ to quit top.

shell-prompt: top
7.9.1 Self-test

1. What is a process?
2. When are processes created?
3. How does the Unix system distinguish between all the processes running at a given moment?
4. What command(s) will show you the processes that are currently running?

7.10 The Unix File system

7.10.1 Unix Files

A Unix file is simply a sequence of bytes (8-bit values) stored on a disk and given a unique name. The bytes in a file may be printable characters such as letters, digits, punctuation symbols, invisible control characters (which cause a printer or terminal to perform actions such as backspacing or scrolling), or other non-character data.

Text vs Binary Files

Files are often classified as either text or binary files. All of the bytes in a text file are interpreted as characters by the programs that read or write the file, while binary files may contain both character and non-character data.

Note that the Unix standard makes no distinction between one file and another based on their contents. Only individual programs care what is inside a file.

Practice Break

Try the following commands:

```
shell-prompt: cat .profile
```

What do you see? The .profile file is a text file, and cat is used here to echo it to the screen.

Now try the following:

```
shell-prompt: cat /bin/ls
```

What do you see? The file /bin/ls is not a text file. It contains binary program code, not characters. The cat command assumes that the file is a text file displays each character terminal. Binary files show up as a lot of garbage, and may even knock your terminal out of whack. If this happens, run the reset command to restore your terminal to its original state. In the rare case that reset does not fix the terminal, you can try running an editor such as vi, which will attempt to reset the terminal when starting or exiting, or simply log out and log back in using a fresh terminal window.

Unix vs. Windows Text Files

While it is the programs that interpret the contents of a file, there are some conventions regarding text file format that all Unix programs follow, so that they can all manipulate the same files. Unfortunately, DOS and Windows programs follow different conventions. Unix programs assume that text files terminate each line with a control character known as a line feed (also known as a newline), which is character 10 in the standard character sets. DOS and Windows programs tend to use both a carriage return (character 13) and a line feed.

To compound the problem, many Unix editors and tools also run on Windows (under Cygwin, for example). As a result, text files may end up with a mixture of line-terminations after being edited on both Windows and Unix.

Some programs are smart enough to properly handle either line termination convention. However, many others will misbehave if they encounter the "wrong" type of line termination.

The dos2unix and unix2dos commands can be used to clean up files that have been transferred between Unix and DOS/Windows. These programs convert text files between the DOS/Windows and Unix standards. If you’ve edited a text file on a non-Unix system, and are now using it on a Unix system, you can clean it up by running:
The *dos2unix* and *unix2dos* commands are not standard with most Unix systems, but they are free programs that can easily be added.

**Caution** Note that *dos2unix* and *unix2dos* should only be used on text files. They should never be used on binary files, since the contents of a binary file are not meant to be interpreted as characters such as line feeds and carriage returns.

### 7.10.2 File system Organization

#### Basic Concepts

A Unix file system contains *files* and *directories*. A file is like a document, and a directory is like a folder that contains documents and/or other directories. The terms "directory" and "folder" are interchangeable, but "directory" is the standard term used in Unix.

#### Note

Unix file systems use case-sensitive file and directory names. I.e., *Temp* is not the same as *temp*, and both can coexist in the same directory.

Mac OS X is the only major Unix system that violates this convention. The standard OS X file system (called HFS) is case-preserving, but not case-sensitive. This means that if you call a file *Temp*, it will remember that the T is capital, but it can also be referred to as *t.emp*, *tEmp*, etc. Only one of these files can exist in a given directory at any one time.

A Unix file system can be visualized as a tree, with each file and directory contained within another directory.

Figure 7.1 shows a small portion of a typical Unix file system. On a real Unix system, there are usually thousands of files and directories. Directories are shown in green and files are in yellow.

![Figure 7.1: Sample of a Unix File system](image)

The one directory that is not contained within any other is known as the *root directory*, whose name under Unix is `/`. There is exactly one root directory on every Unix system. Windows systems, on the other hand, have a root directory for each disk partition such as C: and D:.

The Cygwin compatibility layer works around the separate drive letters of Windows by unifying them under a common parent directory called `/cygdrive`. Hence, for Unix commands run under Cygwin, `/cygdrive/c` is equivalent to c:, `/cygdrive/d` is equivalent to d:, and so on. This allows Cygwin users to traverse multiple Windows drive letters with a single command starting in `/cygdrive`.
Unix file system trees are fairly standardized, but most have some variation. For instance, all Unix systems have a /bin and a /usr/bin, but not all of them have /home or /usr/local.

The root directory is the parent of /bin and /home and an ancestor of all other files and directories.

The /bin and /home directories are subdirectories, or children of /. Likewise, /home/joe and /home/sue are subdirectories of /home, and grandchildren of /.

All of the files in and under /home comprise a subtree of /home.

The children of a directory, all of its children, and so on, are known as descendants of the directory. All files and directories on a Unix system, except /, are descendants of /.

Each user has a home directory, which can be arbitrarily assigned, but is generally a child of /home on many Unix systems. The home directory can be referred to as ~ or ~username in modern Unix shells.

In the example above, /home/joe is the home directory for user joe, and /home/sue is the home directory for user sue. This is the conventional location for home directories on BSD and Linux systems, which are two specific types of Unix. On a Mac OS X system, which is another brand of Unix, Joe’s home directory would be /Users/joe instead of /home/joe. Most of the files owned by ordinary users are either in their home directory or one of its descendants.

**Full Path Names**

The full path name, also known as absolute path name, of a file or directory denotes the complete path from / (the root directory) to the file or directory of interest. For example, the full path name of Sue’s .cshrc file is /home/sue/.cshrc, and the full path name of the ape command is /usr/local/bin/ape.

Note A full path name always begins with '/' or a '~'.

---

**Practice Break**

Try the following commands:

```bash
shell-prompt: ls
shell-prompt: ls /etc
shell-prompt: cat /etc/motd
```

---

**Current Working Directory**

Every Unix process has an attribute called the current working directory, or CWD. This is the directory that the process is currently "in". When you first log into a Unix system, the shell process’s CWD is set to your home directory.

The `pwd` command prints the CWD of the shell process. The `cd` command changes the CWD of the shell process. Running `cd` with no arguments sets the CWD to your home directory.

**Practice Break**

Try the following commands:

```bash
shell-prompt: pwd
shell-prompt: cd /
shell-prompt: pwd
shell-prompt: cd
shell-prompt: pwd
```

---

Some commands, such as `ls`, use the CWD as a default if you don’t provide a directory name on the command line. For example, if the CWD is `/home/joe`, then the following commands are the same:
Relative Path Names

Whereas a full path name denotes the path from / to a file or directory, the relative path name denotes the path from the CWD to a file or directory.

Any path name that does not begin with a '/' or '~' is interpreted as a relative path name. The full path name is then derived by appending the relative path name to the CWD. For example, if the CWD is /etc, then the relative path name motd refers to the full path name /etc/motd.

full path name = CWD + '/' + relative path name

Note Relative path names are handled at the lowest level of the operating system, by the Unix kernel. This means that they can be used anywhere: in shell commands, in C or Fortran programs, etc.

When you run a program from the shell, the new process inherits the CWD from the shell. Hence, you can use relative path names as arguments in any Unix command, and they will use the CWD inherited from the shell. For example, the two cat commands below have the same effect.

```
shell-prompt: cd /etc   # Set shell’s CWD to /etc
shell-prompt: cat motd  # Inherits CWD from shell
shell-prompt: cat /etc/motd
```

Wasting Time

The cd command is one of the most overused Unix commands. Many people use it where it is completely unnecessary and actually results in significantly more typing than needed. Don’t use cd where you could have used the directory with another command. For example, the sequence of commands:

```
shell-prompt: cd /etc
shell-prompt: more hosts
shell-prompt: cd
```

The same effect could have been achieved much more easily using the following single command:

```
shell-prompt: more /etc/hosts
```

Note In almost all cases, full path names and relative path names are interchangeable. You can use either type of path name as a command line argument, or within a program.

Practice Break

Try the following commands:

```
shell-prompt: cd
shell-prompt: pwd
shell-prompt: cd /etc
shell-prompt: pwd
shell-prompt: cat motd
shell-prompt: cat /etc/motd
shell-prompt: cd
shell-prompt: pwd
shell-prompt: cat motd
```

Why does the last command result in an error?
Avoid Full Path Names

The relative path name is potentially much shorter than the full path name. Using relative path names also provides more flexibility.

Suppose you have a project contained in the directory /Users/joe/Thesis on your Mac workstation.

Now suppose you want to work on the same project on a cluster, where there is no /Users directory and you have to store it in /share1/joe/Thesis.

The full path name of every file and directory will be different on the cluster than it is on your Mac. This can cause major problems if you were using full path names in your scripts, programs, and makefiles. Statements like the following will have to be changed in order to run the program on a different computer.

```
infile = fopen("/Users/joe/Thesis/Inputs/input1.txt", "r");
```

```
sort /Users/joe/Thesis/Inputs/names.txt
```

No program should ever have to altered just to run it on a different computer.

While the full path names change when you move the Thesis directory, the path names relative to the Thesis directory remain the same. For this reason, full path names should be avoided unless absolutely necessary.

The statements below will work on any computer as long as the program or script is running with Thesis as the current working directory. It does not matter where the Thesis directory is located, so long as the Inputs directory is its child.

```
infile = fopen("Inputs/input1.txt", "r");
```

```
sort Inputs/names.txt
```

Special Directory Names

In addition to full path names and relative path names, there are a few special symbols for directories that are commonly referenced:

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Refers to</th>
</tr>
</thead>
<tbody>
<tr>
<td>.</td>
<td>The current working directory</td>
</tr>
<tr>
<td>..</td>
<td>The parent of the current working directory</td>
</tr>
<tr>
<td>~</td>
<td>Your home directory</td>
</tr>
<tr>
<td>~user</td>
<td>user’s home directory</td>
</tr>
</tbody>
</table>

Table 7.4: Special Directory Symbols

Practice Break

Try the following commands:

```
shell-prompt: cd
shell-prompt: pwd
shell-prompt: ls
shell-prompt: ls .
shell-prompt: cp /etc/motd .
shell-prompt: cat motd
shell-prompt: cat ./motd
shell-prompt: ls ~
shell-prompt: ls /home
shell-prompt: ls ..
shell-prompt: cd ..
shell-prompt: ls
shell-prompt: ls ~
shell-prompt: cd
```
7.10.3 Ownership and Permissions

Overview

Every file and directory on a Unix system has inherent access control features based on a simple scheme:

- Every file and directory has an individual owner and group owner.
- There are 3 types of permissions which are controlled separately from each other:
  - Read
  - Write (create or modify)
  - Execute (e.g. run a file if it’s a program)
- Read, write, and execute permissions can be granted or denied separately for each of the following:
  - The individual owner
  - The group owner
  - All users on the system (a hypothetical group known as "world")

Execute permissions on a file mean that the file can be executed as a script or a program by typing its name. It does not mean that the file actually contains a script or a program: It is up to the owner of the file to set the execute permissions appropriately for each file.

Execute permissions on a directory mean that users in the category can `cd` into it. Users only need read permissions on a directory to list it or access a file within it, but they need execute permissions to make it the current working directory of their processes.

Unix systems provide this access using 9 on/off switches (bits) associated with each file.

Viewing Permissions

If you do a long listing of a file or directory, you will see the ownership and permissions:

```
shell-prompt: ls -l
drwx------ 2 joe users 512 Aug 7 07:52 Desktop/
drwxr-x--- 39 joe users 1536 Aug 9 22:21 Documents/
drwxr-xr-x 2 joe users 512 Aug 9 22:25 Downloads/
-rw-r--r-- 1 joe users 82118 Aug 2 09:47 bootcamp.pdf
```

The leftmost column shows the type of object and the permissions for each user category.
A ‘-’ in the leftmost character means a regular file, ’d’ means a directory, ’l’ means a link. etc. Running `man ls` will reveal all the codes.

The next three characters are, in order, read, write and execute permissions for the owner.

The next three after that are permissions for members of the owning group.

The next three are permissions for world.

A ‘-’ in a permission bit column means that the permission is denied for that user or set of users and an ’r’, ’w’, or ’x’ means that it is enabled.

The next two columns show the individual and group ownership of the file or directory. The other columns show the size, the date and time it was last modified, and name. In addition to the ’d’ in the first column, directory names are followed by a ’/’.

You can see above that Joe’s Desktop directory is readable, writable, and executable for Joe, and completely inaccessible to everyone else.

Joe’s Documents directory is readable, writable and executable for Joe, and readable and executable for members of the group "users". Users not in the group "users" cannot access the Documents directory at all.

Joe’s Downloads directory is readable and executable to anyone who can log into the system.

The file bootcamp.pdf is readable by the world, but only writable by Joe. It is not executable by anyone, which makes sense because a PDF file is not a program.
Setting Permissions

Users cannot change individual ownership on a file, since this would allow them to subvert disk quotas by placing their files under someone else’s name. Only the superuser can change the individual ownership of a file or directory.

Users can change the group ownership of a file to any group that they belong to using the chgrp command:

```
shell-prompt: chgrp group path [path ...]
```

All sharing of files on Unix systems is done by controlling group ownership and file permissions.

File permissions are changed using the chmod command:

```
shell-prompt: chmod permission-specification path [path ...]
```

The permission specification has a symbolic form, and a raw form, which is an octal number.

The basic symbolic form consists of any of the three user categories ‘u’ (user/owner), ‘g’ (group), and ‘o’ (other/world) followed by a ‘+’ (enable) or ‘-’ (disable), and finally one of the three permissions ‘r’, ‘w’, or ‘x’.

Add read and execute permissions for group and world on the Documents directory:

```
shell-prompt: chmod go+rx Documents
```

Disable all permissions for world on the Documents directory:

```
shell-prompt: chmod o-rwx Documents
```

Disable write permission for everyone, including the owner, on bootcamp.pdf. (This can be used to reduce the chances of accidentally deleting an important file.)

```
shell-prompt: chmod ugo-w bootcamp.pdf
```

Run man chmod for additional information.

The raw form for permissions uses a 3-digit octal number to represent the 9 permission bits. This is a quick and convenient method for computer nerds who can do octal/binary conversions in their head.

```
shell-prompt: chmod 644 bootcamp.pdf # 644 = 110100100 = rw-r--r--
shell-prompt: chmod 750 Documents # 750 = 111101000 = rwxr-x---
```

![Caution] NEVER make any file or directory world-writable. Doing so allows any other user to modify it, which is a serious security risk. A malicious user could replace use this to install a Trojan Horse program under your name, for example.

Practice Break

Try the following commands, and try to predict the output of each ls before you run it.

```
shell-prompt: touch testfile
shell-prompt: ls -l
shell-prompt: chmod go-rwx testfile
shell-prompt: ls -l
shell-prompt: chmod o+rw testfile
shell-prompt: ls -l
shell-prompt: chmod g+rwx testfile
shell-prompt: ls -l
shell-prompt: rm testfile
```

Now set permissions on testfile so that it is readable, writable, and executable by you, only readable by the group, and inaccessible to everyone else.
7.10.4 Advanced: Unix File System Comparison

Windows file systems become fragmented over time as file are created and removed. Windows users should therefore run a defragmentation tool periodically to improve disk performance.

Unix file systems, in contrast, do continuous defragmentation, so performance will not degrade significantly over time.

Overwrite performance on some file systems is slower than initial write. Hence, removing files before overwriting them may help program run times.

Most Unix systems offer multiple choices for file systems. Most modern file systems use journaling, in which data that is critical to maintaining file system integrity in the event of a system crash is written to the disk immediately instead of waiting in a memory buffer.

To save time, this data is queued to a special area on the disk known called the journal. Writing to a journal is faster than saving the data in it’s final location, since it requires fewer disk head movements.

Journaling reduces write performance, since data is first written to a journal and later moved to its final location. This takes more time and more disk head movements than storing data in a memory buffer until it is written to its final location. However, the performance penalty is marginal if done intelligently. All modern Unix file systems use advanced journaling methods to minimize the performance hit and disk wear.

Popular file systems:

• EXT is the most commonly used file system on Linux systems. EXT3 was the first to including journaling, basically as a feature added to EXT2. EXT2 was notorious for incredibly slow file system checks and repairs. The journaling features added by EXT3 greatly reduced the need for repairs, but EXT3 is not the best performer overall and is also hard on disks due to excessive head movements.

EXT4 represents a vast improvement over EXT3 due to major redesign of key components. Performance and reliability are solid.

• HFS is the file system used by Mac OS X. Features and performance are generally positive. One potential problem for Unix users is the lack of true case-sensitivity. HFS is case-preserving, but not case-sensitive. This means that if you create a file named "Tempfile", the "T" will be remembered as a capital. However, it is not distinguished from a lower-case ‘t’, so the file may be referred to as "tempfile". Also, you cannot have two files in the same directory called "Tempfile" and "tempfile", because these two file names are considered the same.

• UFS (Unix File System) evolved from the original Unix system 7 file system and is now used by most BSD systems as well as some commercial systems such as SunOS/Solaris and HP-UX.

FreeBSD’s UFS2 includes a unique feature called soft updates, which protects file system integrity in the event of a system crash without using a journal. This allows UFS2 to exhibit better write performance and less disk wear.

• XFS is a file system developed by SGI for it’s commercial IRIX operating system during the 1990s, which were popular for high-end graphics. SGI IRIX machines were used to develop and featured in the movie Jurassic Park.

XFS has been fully integrated into Linux and is now used as an alternative to EXT4 where high performance and very large partitions are desired.

• ZFS is a unique combination of a file system combined with a volume manager, developed by Sun Microsystems.

ZFS is widely regarded as the most advanced file system to date. It features variable-sized partitions, software RAID that generally outperforms hardware RAID systems, and many other advanced features such as compression and encryption.

ZFS has been fully integrated into FreeBSD and is now the default file system for high-end FreeBSD servers as well as the TrueOS desktop system. ZFS does require a lot of RAM, however, so UFS2 is still a better choice for low-end hardware such as net books and embedded FreeBSD systems.

7.10.5 Advanced: Network File System

Network File System, or NFS, is a standard Unix network protocol that allows disk partitions on one Unix to be directly accessed from other computers. In concept, NFS is similar to Apple’s AFS and Microsoft’s SMB/CIFS.
Access to files across an NFS link is generally somewhat slower than local disk access, due to the overhead of network communication. Speed may be limited either by the local disk performance on the NFS server or by the bandwidth of the network. For example, of an NFS server has a RAID that can deliver 500 megabytes per second locally and a 1 gigabit (~100 megabyte per second) network, then the disk performance seen by NFS clients will be limited by the network to about 100 megabytes per second.

Unix systems also allow other computers to access their disks using non-Unix protocols like AFS and SMB/CIFS if necessary. For example, Samba is an open source implementation of the SMB/CIFS protocol that allows Windows computers to access data on Unix disks.

### 7.10.6 Self-test

1. What is a Unix file?
2. Explain the difference between a text file and a binary file.
3. Do Unix operating systems distinguish between text files and binary files? Explain.
4. Does the Unix standard include a convention on the format of text files?
5. Are Unix text files the same as Windows (and DOS) text files? Explain.
6. How can text files be converted between Windows and Unix conventional formats?
7. What is a directory? Does it go by any other names?
8. How are files and directories organized in a Unix file system?
9. What are some of the conventional directories in the Unix file system organization?
10. What is the root directory?
11. What is a parent directory?
12. What is a sibling directory?
13. What is a child directory?
14. What is a subdirectory?
15. What is a descendant directory?
16. What is an ancestor directory?
17. What is a home directory?
18. What is a subtree?
19. What is a full or absolute path name?
20. What is the current working directory? What is a current working directory a property of?
21. What is a relative path name? How can you convert a relative path name to a full path name?
22. Why should full path names be avoided?
23. How can you determine the current working directory of your shell process?
24. How can you change the current working directory of your shell process to each of the following?
   (a) Your home directory.
   (b) /etc
   (c) The directory Program1, which is a subdirectory of Programs, which is a subdirectory of the current working directory.
25. Show the simplest Unix command to view each of the following files?
   (a) /etc/hosts
   (b) A file called .cshrc in the current working directory.
   (c) A file called .cshrc in your home directory, regardless of what the current working directory is.
   (d) A file called .cshrc in the home directory of a user with user name "bacon".
   (e) A file called readme.txt in the parent directory of the current working directory.

26. How can you change the group ownership of the file .cshrc in your home directory to the group "smithlab"?

27. How can you change the individual ownership of the file .cshrc in your home directory to your friend with user name Bob? Explain.

28. How can you change the permissions on the file .cshrc in your home directory so that only you can modify it, members of the group can read and execute it but not modify it, and anyone else can read it but not modify or execute it?

29. How can you see the ownership and permissions on all the files in /etc? In the current working directory?

7.11 Unix Commands and the Shell

**Before You Begin** You should have a basic understanding of Unix processes, files, and directories. These topics are covered in Section 7.9 and Section 7.10.

Unix commands fall into one of two categories:

- **Internal commands** are part of the shell. No new process is created when you execute an internal command. The shell simply carries out the execution of internal commands by itself.

- **External commands** are programs separate from the shell. The command name of an external command is actually the name of an **executable file**, i.e. a file containing the program or script. For example, when you run the `ls` command, you are executing the program contained in the file `/bin/ls`.

When you run an external command, the shell locates the program file, loads the program into memory, and creates a new (child) process to execute the program. The shell then normally waits for the child process to end before prompting you for the next command.

### 7.11.1 Internal Commands

Commands are implemented internally only when it is necessary, or when there is a substantial benefit. If all commands were part of the shell, the shell would be enormous and would require too much memory.

An example is the `cd` command, which changes the CWD of the shell process. The `cd` command cannot be implemented as an external command, since the CWD is a property of the process. If the `cd` command were external, it would run as a child process of the shell. Altering the CWD of a child process does not affect the CWD of the parent process. Each has its own separate CWD.

Likewise, any command that alters the state of the shell process must be implemented as an internal command.

A command might also be implemented internally simply because it’s trivial to do so, and it saves the overhead of loading and running an external command. When the work done by a command is very simple, it might take more resources to load an external program than it does to actually run it. In these cases, it makes more sense to implement it as part of the shell.
7.11.2 External Commands

The executable files containing external commands are kept in certain directories, most of which are called `bin` (short for binary, since most executable files are binary files). The most essential commands that are common to most Unix systems are kept in `/bin` and `/usr/bin`. The location of optional add-on commands varies, but a typical location is `/usr/local/bin`.

The list of directories that are searched when looking for external commands is kept in an environment variable called `PATH`. The environment is discussed in more detail in Section 7.15.

### Practice Break

1. Use `which` to find out whether the following commands are internal or external.

   ```
   shell-prompt: which cd
   shell-prompt: which cp
   shell-prompt: which exit
   shell-prompt: which ls
   shell-prompt: which pwd
   ```

2. Find out where your external commands are stored by running `echo $PATH`.

3. Use `ls` to find out what commands are located in `/bin` and `/usr/bin`.

7.11.3 Getting Help

In the olden days before Unix, when programmers wanted to look up a command or function, they would have to get out of their chair and walk somewhere to pick up a typically ring-bound manual to flip through.

The Unix designers saw this as a waste of time. They thought, wouldn’t it be nice if we could sit in the same chair for ten hours straight without ever taking our eyes off the monitor or our hands off the keyboard?

And so, online documentation was born. On Unix systems, all common Unix commands are documented in detail on the Unix system itself, and the documentation is accessible via the command line (you do not need a GUI to view it). Whenever you want to know more about a particular Unix command, you can find out by typing `man command-name`. For example, to learn all about the `ls` command, type:

```
shell-prompt: man ls
```

The `man` covers virtually every common command, as well as other topics. It even covers itself:

```
shell-prompt: man man
```

The `man` command displays a nicely formatted document known as a `man page`. It uses a file viewing program called `more`, which can be used to browse through text files very quickly. Table 7.5 shows the most common keystrokes used to navigate a man page. For complete information on navigation, run:

```
shell-prompt: man more
```

<table>
<thead>
<tr>
<th>Key</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>\h</td>
<td>Show key commands</td>
</tr>
<tr>
<td>Space bar</td>
<td>Forward one page</td>
</tr>
<tr>
<td>Enter/Return</td>
<td>Forward one line</td>
</tr>
<tr>
<td>\b</td>
<td>Back one page</td>
</tr>
<tr>
<td>/</td>
<td>Search</td>
</tr>
</tbody>
</table>

Table 7.5: Common hot keys in `more`
Man pages include a number of standard sections, such as SYNOPSIS, DESCRIPTION, and SEE ALSO, which helps you identify other commands that might be of use.

Man pages do not always make good tutorials. Sometimes they contain too much detail, and they are often not well-written for novice users. If you’re learning a new command for the first time, you might want to consult a Unix book or the WEB. The man pages will provide the most detailed and complete reference information on most commands, however.

The apropos command is used to search the man page headings for a given topic. It is equivalent to man -k. For example, to find out what man pages exist regarding Fortran, we might try the following:

```
shell-prompt: apropos fortran
```

or

```
shell-prompt: man -k fortran
```

The whatis is similar to apropos in that it lists short descriptions of commands. However, whatis only lists those commands with the search string in their name or short description, whereas apropos attempts to list everything related to the string.

The info command is an alternative to man that uses a non-graphical hypertext system instead of flat files. This allows the user to navigate extensive documentation more efficiently. The info command has a fairly high learning curve, but it is very powerful, and is often the best option for documentation on a given topic. Some open source software ships documentation in info format and provides a man page (converted from the info files) that actually has less information in it.

```
shell-prompt: info gcc
```

---

**Practice Break**

1. Find out how to display a '/' after each directory name and a '*' after each executable file when running `ls`.

2. Use `apropos` to find out what Unix commands to use with bzip files.

---

### 7.11.4 A Basic Set of Unix Commands

Most Unix commands have short names which are abbreviations or acronyms for what they do. (pwd = print working directory, cd = change directory, ls = list, ...) Unix was originally designed for people with good memories and poor typing skills.

Some of the most commonly used Unix commands are described below.

---

**Note** This section is meant to serve as a quick reference, and to inform new readers about which commands they should learn. There is much more to know about these commands than we can cover here. For full details about any of the commands described here, consult the man pages, info pages, or the WEB.

---

This section uses the same notation conventions as the Unix man pages:

- Optional arguments are shown inside [].
- The pipe symbol (|) between two items means one or the other.
- An ellipses (...) means optionally more of the same.
- "file" means a filename is required and a directory name is not allowed. "directory" means a directory name is required, and a filename is not allowed. "path" means either a filename or directory name is acceptable.
File and Directory Management

**cp** copies one or more files.

```
shell-prompt: cp source-file destination-file
shell-prompt: cp source-file [source-file ...] destination-directory
```

If there is only one source filename, then destination can be either a filename or a directory. If there are multiple source files, then destination must be a directory. If destination is a filename, and the file exists, it will be overwritten.

```
shell-prompt: cp file file.bak       # Make a backup copy
shell-prompt: cp file file.bak ~    # Copy files to home directory
```

**ls** lists files in CWD or a specified file or directory.

```
shell-prompt: ls [path ...]
shell-prompt: ls       # List CWD
shell-prompt: ls /etc  # List /etc directory
```

**mv** moves or renames files or directories.

```
shell-prompt: mv source destination
shell-prompt: mv source [source ...] destination-directory
```

If multiple sources are given, destination must be a directory.

```
shell-prompt: mv progl.c Programs
```

**ln** link files or directories.

```
shell-prompt: ln source-file destination-file
shell-prompt: ln -s source destination
```

The **ln** command creates another path name for the same file. Both names refer to the same file, and changes made through one appear in the other. Without `-s`, a standard directory entry, known as a **hard link** is created. In this case, source and destination must be on the same partition. (The **df** will list partitions and their location within the directory tree.) With `-s`, a **symbolic link** is created. A symbolic link is not a standard directory entry, but a pointer to the source path name. Only symbolic links can be used for directories, and symbolic links to not have to be on the same partition as the source.

```
shell-prompt: ln -s /etc/motd ~       # Make a convenient link to motd
```

**rm** removes one or more files.

```
shell-prompt: rm file [file ...]
shell-prompt: rm temp.txt core a.out
```

---

**Caution** Removing files with **rm** is not like dragging them to the trash. Once files are removed by **rm**, they cannot be recovered.

**mkdir** creates one or more directories.

```
shell-prompt: mkdir [-p] path name [path name ...]
```
The `-p` flag indicates that `mkdir` should attempt to create any parent directories in the path that don’t already exist. If not used, `mkdir` will fail unless all but the last component of the path exist.

```bash
shell-prompt: mkdir Programs
shell-prompt: mkdir -p Programs/C/MPI
```

`rmdir` removes one or more empty directories.

```bash
shell-prompt: rmdir directory [directory ...]
```

`rmdir` will fail if a directory is not completely empty. You may also need to check for hidden files using `ls -a directory`.

```bash
shell-prompt: rmdir Programs/C/MPI
```

`find` locates files within a subtree using a wide variety of possible criteria.

```bash
shell-prompt: find start-directory criteria [action]
```

`find` is a very powerful and complex command that can be used to not only find files, but run commands on the files matching the search criteria.

```bash
shell-prompt: find . -name core # List cores
shell-prompt: find . -name core -exec rm '{}' \; # Remove cores
```

`df` shows the free disk space on all currently mounted partitions.

```bash
shell-prompt: df
```

`du` reports the disk usage of a directory and everything under it.

```bash
shell-prompt: du [-s] [-h] path
```

The `-s` (summary) flag suppresses output about each file in the subtree, so that only the total disk usage of the directory is shown.

```bash
shell-prompt: du -sh Programs
```

### Shell Internal Commands

As mentioned previously, internal commands are part of the shell, and serve to control the shell itself. Below are some of the most common internal commands.

`cd` changes the current working directory of the shell process. It is described in more detail in Section 7.10.2.

```bash
shell-prompt: cd [directory]
```

The `pwd` command prints the CWD of the shell process. It is described in detail in Section 7.10.21. You can use `pwd` like a Unix file system GPS, to get your bearing when you’re lost.

```bash
shell-prompt: pwd
```

`pushd` changes CWD and saves the old CWD on a stack so that we can easily return.

```bash
shell-prompt: pushd directory
```

Users often encounter the need to temporarily go to another directory, run a few commands, and then come back to the current directory.

The `pushd` command is a very useful alternative to `cd` that helps in this situation. It performs the same operation as `cd`, but it records the starting CWD by adding it to the top of a stack of CWDs. You can then return to where the last `pushd` command was invoked using `popd`. This saves you from having to retype the path name of the directory you want to return to. Not all shells support `pushd` and `popd`, but the ones you are likely to use for a login session do.

---

1`pwd` is actually an external command, but we cover it here since it relates to the CWD of the shell process. (recall that new processes inherit the CWD of the shell process, so `pwd` need not be internal.)
**Practice Break**

Try the following sequence of commands:

```
shell-prompt: pwd # Check starting point
shell-prompt: pushd /etc
shell-prompt: more motd
shell-prompt: ls
shell-prompt: popd # Back to /etc
shell-prompt: pwd # Back to starting point
```

**exit** terminates the shell process.

```
shell-prompt: exit
```

This is the most reliable way to exit a shell. In some situations you could also type `logout` or simply press Ctrl+d, but these alternatives will not work for every shell process.

**Text File Processing**

**cat** echoes the contents of one or more text files.

```
shell-prompt: cat file [file ...]
```

```
shell-prompt: cat /etc/motd
```

**more** shows the contents of one or more text files interactively.

```
shell-prompt: more file [file ...]
```

```
shell-prompt: more prog1.c
```

**head** shows the top N lines of one or more text files.

```
shell-prompt: head -n # file [file ...]
```

If a flag consisting of a - followed by an integer number N is given, the top N lines are shown instead of the default of 10.

```
shell-prompt: head -n 5 prog1.c
```

**tail** shows the bottom N lines of one or more text files.

```
shell-prompt: tail -n # file [file ...]
```

Tail is especially useful for viewing the end of a large file that would be cumbersome to view with **more**.

If a flag consisting of a - followed by an integer number N is given, the bottom N lines are shown instead of the default of 10.

```
shell-prompt: tail -n 5 output.txt
```

**grep** shows lines in one or more text files that match a given *regular expression*.

```
shell-prompt: grep regular-expression file [file ...]
```
The regular expression is most often a simple string, but can represent patterns as described by `man re_format`.

Show all lines containing the string "printf" in progl.c.

```bash
shell-prompt: grep printf progl.c
```

Show all lines containing the variable names in progl.c. (Variable names begin with a letter or underscore and may contain letters, underscores, or digits after that.)

```bash
shell-prompt: grep '[a-zA-Z_]\[a-zA-Z0-9_]*' progl.c
```

The `diff` command shows the differences between two text files. This is most useful for comparing two versions of the same file to see what has changed. Also see `ediff`, a specialized version of `diff`, for comparing C source code.

```bash
shell-prompt: diff -u input1.txt input2.txt
```

### Text Editors

There are more text editors available for Unix systems than any one person is aware of. Some are terminal-based, some are graphical, and some have both types of interfaces.

All Unix systems support running graphical programs from remote locations, but most graphical programs require a fast connection (10 megabits/sec) or more to function tolerably.

Knowing how to use a terminal-based text editor is therefore a very good idea, so that you’re prepared to work over a slow connection if necessary. Some of the more common terminal-based editors are described below.

- **vi** (visual editor) is the standard text editor for all Unix systems. Most users either love or hate the vi interface, but it’s a good editor to know since it is standard on every Unix system.

- **nano** is an extremely simplistic text editor that is ideal for beginners. It is a rewrite of the **pico** editor, which is known to have many bugs and security issues. Neither editor is standard on Unix systems, but both are free and easy to install. These editors entail little or no learning curve, but are not sophisticated enough for extensive programming or scripting.

- **emacs** (Edit MACroS) is a more sophisticated editor used by many programmers. It is known for being hard to learn, but very powerful. It is not standard on most Unix systems, but is free and easy to install.

- **ape** is a menu-driven, user-friendly IDE (integrated development environment), i.e. programmer’s editor. It has an interface similar to PC and Mac programs, but works on a standard Unix terminal. It is not standard on most Unix systems, but is free and easy to install. **ape** has a small learning curve, and advanced features to make programming much faster.

### Networking

- **hostname** prints the network name of the machine.

  ```bash
  shell-prompt: hostname
  ```

  This is often useful when you are working on multiple Unix machines at the same time (e.g. via `ssh`), and forgot which window applies to each machine.

- **ssh** is used to remotely log into another machine on the network and start a shell.

  ```bash
  ssh [name@]hostname
  
  shell-prompt: ssh joe@login.peregrine.hpc.uwm.edu
  ```

- **sftp** is used to remotely log into another machine on the network and transfer files to or from it.

  ```bash
  shell-prompt: sftp [name@]host
  
  shell-prompt: sftp joe@data.peregrine.hpc.uwm.edu
  ```
rsync is used to synchronize two directories either on the same machine or on different machines.

```bash
shell-prompt: rsync [flags] [[name@]host:]path [[name@]host:]path
```

rsync compares the contents of the two source and destination directories and transfers only the differences. Hence, it can save an enormous amount of time when you make small changes to a large project and need to synchronize another copy of the project.

```bash
shell-prompt: rsync -av Project joe@data.peregrine.hpc.uwm.edu:
```

Identity and Access Management

`passwd` changes your password. It asks for your old password once, and the new one twice (to ensure that you don’t accidentally set your password to something you don’t know because your finger slipped). Unlike many graphical password programs, `passwd` does not echo anything for each character typed. (Even showing the length of your password is a bad idea from a security standpoint.)

```bash
shell-prompt: passwd
```

Terminal Control

`clear` clears your terminal screen (assuming the TERM variable is properly set).

```bash
shell-prompt: clear
```

`reset` resets your terminal to an initial state. This is useful when your terminal has been corrupted by bad output, such as when attempting to view a binary file.

Terminals are controlled by magic sequences, sequences of invisible control characters sent from the host computer to the terminal amid the normal output. Magic sequences move the cursor, change the color, change the international character set, etc. Binary files contain random data that sometimes by chance contain magic sequences that could alter the mode of your terminal. If this happens, running `reset` will usually correct the problem. If not, you will need to log out and log back in.

```bash
shell-prompt: reset
```

7.11.5 Self-test

1. What is an internal command?
2. What is an external command?
3. What kinds of commands must be implemented as internal commands?
4. How can you quickly view detailed information on a Unix command?
5. How can you identify Unix commands related to a topic? (Describe two methods.)
6. Show the simplest Unix command to accomplish each of the following in order:
   (a) List the files in `/usr/local/share`.
   (b) Make your home directory the CWD.
   (c) Copy the file `/etc/hosts` to your home directory.
   (d) Rename the file `~/hosts` to `~/hosts.bak`.
   (e) Create a subdirectory called `~/Temp`.
   (f) Make `~/Temp` the CWD.
   (g) Copy the file `~/hosts.bak` to `~/Temp`. 
(h) Create a hard link to the file ~/Temp/hosts.bak called ~/hosts.bak.temp.
(i) Create a link to the directory /usr/local/share in your home directory.
(j) Make your home directory the CWD.
(k) Remove the entire subtree in ~/Temp and the files ~/hosts.bak and ~/hosts.bak.temp.
(l) Show how much space is used by the directory /etc.

7. Show the simplest Unix command to accomplish each of the following:
(a) Change the current working directory of your shell process to /etc, remembering the previous current working directory on the directory stack.
(b) Return to the previous current working directory on the directory stack.
(c) Terminate the shell.

8. Show the simplest Unix command to accomplish each of the following:
(a) Show the contents of the text file /etc/motd a page at a time.
(b) Show the first 5 lines of /etc/motd.
(c) Show the last 8 lines of /etc/motd.
(d) Show lines in /etc/group and /etc/passwd containing your username.
(e) Edit the text file ./prog1.c.

9. Show the simplest Unix command to accomplish each of the following:
(a) Show the network name (host name) of the computer running the shell.
(b) Remotely log into "login.peregrine.hpc.uwm.edu" and start a shell.
(c) Remotely log into "data.peregrine.hpc.uwm.edu" for the purpose of transferring files.
(d) Synchronize the folder ~/Programs/Prog1 on login.peregrine.hpc.uwm.edu to ./Prog1, transferring only the differences.
(e) Clear the terminal screen.
(f) Restore functionality to a terminal window that’s in a funk.

7.12 Unix Command Quick Reference

Table 7.6 provides a quick reference for looking up common Unix commands. For details on any of these commands, run man command (or info command on some systems).

7.13 POSIX and Extensions

Unix-compatible systems generally conform to standards published by the International Organization for Standardization (ISO), the Open Group, and the IEEE Computer Society.

The primary standard used for this purpose is POSIX, the portable operating system standard based on Unix.

Programs and commands that conform to the POSIX standard should work on any Unix system. Therefore, developing your programs and scripts according to POSIX will prevent problems and wasted time.

Nevertheless, many common Unix programs have been enhanced beyond the POSIX standard to provide useful features. Fortunately, most such programs are open source and can therefore be easily installed on most Unix systems.

Features that do not conform to the POSIX standard are known as extensions. Extensions are often described according to their source, e.g. BSD extensions or GNU extensions.
### Synopsis

<table>
<thead>
<tr>
<th>Command</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>`ls [file</td>
<td>directory]`</td>
</tr>
<tr>
<td><code>cp source-file destination-file</code></td>
<td>Copy a file</td>
</tr>
<tr>
<td><code>cp source-file [source-file ...] directory</code></td>
<td>Copy multiple files to a directory</td>
</tr>
<tr>
<td><code>mv source-file destination-file</code></td>
<td>Rename a file</td>
</tr>
<tr>
<td><code>mv source-file [source-file ...] directory</code></td>
<td>Move multiple files to a directory</td>
</tr>
<tr>
<td><code>ln source-file destination-file</code></td>
<td>Create another name for the same file. (source and destination must be in the same file system)</td>
</tr>
<tr>
<td><code>ln -s source destination</code></td>
<td>Create a symbolic link to a file or directory</td>
</tr>
<tr>
<td><code>rm file [file ...]</code></td>
<td>Remove one or more files</td>
</tr>
<tr>
<td><code>rm -r directory</code></td>
<td>Recursively remove a directory and all of its contents</td>
</tr>
<tr>
<td><code>mkdir directory</code></td>
<td>Create a directory</td>
</tr>
<tr>
<td><code>rmdir directory</code></td>
<td>Remove a directory (the directory must be empty)</td>
</tr>
<tr>
<td><code>find start-directory criteria</code></td>
<td>Find files/directories based on flexible criteria</td>
</tr>
<tr>
<td><code>make</code></td>
<td>Rebuild a file based on one or more other files</td>
</tr>
<tr>
<td><code>od/hexdump</code></td>
<td>Show the contents of a file in octal/hexadecimal</td>
</tr>
<tr>
<td><code>awk</code></td>
<td>Process tabular data from a text file</td>
</tr>
<tr>
<td><code>sed</code></td>
<td>Stream editor. Echo files, making changes to contents.</td>
</tr>
<tr>
<td><code>sort</code></td>
<td>Sort text files based on flexible criteria</td>
</tr>
<tr>
<td><code>uniq</code></td>
<td>Echo files, eliminating adjacent duplicate lines.</td>
</tr>
<tr>
<td><code>diff</code></td>
<td>Show differences between text files.</td>
</tr>
<tr>
<td><code>cmp</code></td>
<td>Detect differences between binary files.</td>
</tr>
<tr>
<td><code>cdiff</code></td>
<td>Show differences between C programs.</td>
</tr>
<tr>
<td><code>cut</code></td>
<td>Extract substrings from text.</td>
</tr>
<tr>
<td><code>m4</code></td>
<td>Process text files containing m4 mark-up.</td>
</tr>
<tr>
<td><code>chfn</code></td>
<td>Change finger info (personal identity).</td>
</tr>
<tr>
<td><code>chsh</code></td>
<td>Change login shell.</td>
</tr>
<tr>
<td><code>su</code></td>
<td>Substitute user.</td>
</tr>
<tr>
<td><code>cc/gcc/cc</code></td>
<td>Compile C programs.</td>
</tr>
<tr>
<td><code>f77/f90/gfortran/ifort</code></td>
<td>Compile Fortran programs.</td>
</tr>
<tr>
<td><code>ar</code></td>
<td>Create static object libraries.</td>
</tr>
<tr>
<td><code>indent</code></td>
<td>Beautify C programs.</td>
</tr>
<tr>
<td><code>astyle</code></td>
<td>Beautify C, C++, C#, and Java programs.</td>
</tr>
<tr>
<td><code>tar</code></td>
<td>Pack a directory tree into a single file.</td>
</tr>
<tr>
<td><code>gzip</code></td>
<td>Compress files.</td>
</tr>
<tr>
<td><code>gunzip</code></td>
<td>Uncompress gzipped files.</td>
</tr>
<tr>
<td><code>bzip2</code></td>
<td>Compress files better (and slower).</td>
</tr>
<tr>
<td><code>bunzip2</code></td>
<td>Uncompress bzipped files.</td>
</tr>
<tr>
<td><code>zcat/zmore/zgrep/bzcat/bzmore/bzgrep</code></td>
<td>Process compressed files.</td>
</tr>
<tr>
<td><code>exec command</code></td>
<td>Replace shell process with command.</td>
</tr>
<tr>
<td><code>date</code></td>
<td>Show the current date and time.</td>
</tr>
<tr>
<td><code>cal</code></td>
<td>Print a calendar for any month of any year.</td>
</tr>
<tr>
<td><code>bc</code></td>
<td>Unlimited precision calculator.</td>
</tr>
<tr>
<td><code>printenv</code></td>
<td>Print environment variables.</td>
</tr>
</tbody>
</table>

Table 7.6: Unix Commands
Many base commands such as awk, make, and sed, may contain extensions that depend on the specific operating system. For example, BSD systems use the BSD versions of awk, make, and sed, which contain BSD extensions, while GNU/Linux systems use the GNU versions of awk, make, and sed, which contain GNU extensions.

When installing GNU software on BSD systems, the GNU version of the command is often prefixed with a 'g', to distinguish it from the native BSD command. For example, on FreeBSD, "make" and "awk" are the BSD implementations and "gmake" and "gawk" would be the GNU implementations. Likewise, on GNU/Linux systems, BSD commands would generally be prefixed with a 'b' or 'bsd'. The "make" and "tar" commands would refer to GNU versions and BSD make would be run as "bmake" and BSD tar as "bsdtar".

All of them will support POSIX features, so if you use only POSIX features, they will all behave the same. If you want to use GNU or other extensions, it’s generally best to use the extended command name, e.g. gawk instead of awk.

<table>
<thead>
<tr>
<th>Program</th>
<th>Sample of extensions</th>
</tr>
</thead>
<tbody>
<tr>
<td>BSD Tar</td>
<td>Support for extracting ISO and Apple DMG files</td>
</tr>
<tr>
<td>GNU Make</td>
<td>Various &quot;shortcut&quot; rules for compiling multiple source files</td>
</tr>
<tr>
<td>GNU Awk</td>
<td>Additional built-in functions</td>
</tr>
</tbody>
</table>

Table 7.7: Common Extensions

### 7.14 File Transfer

Many users will need to transfer data between other computers and a remote Unix system. For example, users of a shared research computer running Unix will need to transfer input data from their computer to the Unix machine, run the research programs, and finally transfer results back to their computer. There are many software tools available to accomplish this. Some of the more convenient tools are described below.

#### 7.14.1 File Transfers from Unix

For Unix (including Mac and Cygwin) users, the recommended method for transferring files is the `rsync` command. The rsync command is a simple but intelligent tool that makes it easy to synchronize two directories on the same machine or on different machines across a network. Rsync is free software and part of the base installation of many Unix systems including Mac OS X. On Cygwin, you can easily add the rsync package using the Cygwin Setup utility.

Rsync has two major advantages over other file transfer programs:

- If you have transferred the directory before, and only want to update it, rsync will automatically determine the differences between the two copies and only transfer what is necessary. When conducting research that generates large amounts of data, this can save an enormous amount of time.
- If a transfer fails for any reason (which is more likely for large transfers), rsync’s inherent ability to determine the differences between two copies allows it to resume from where it left off. Simply run the exact same rsync command again, and the transfer will resume.

The rsync command can either push (send) files from the local machine to a remote machine, or pull (retrieve) files from a remote machine to the local machine. The command syntax is basically the same in both cases. It’s just a matter of how you specify the source and destination for the transfer.

The rsync command has many options, but the most typical usage is to create an exact copy of a directory on a remote system. The general rsync command to push files to another host would be:

```bash
shell-prompt: rsync -av --delete source-path [username@]hostname:[destination-path]
```
Example 7.2 Pushing data with rsync
The following command synchronizes the directory Project from the local machine to ~joeuser/Data/Project on Peregrine:

```
shell-prompt: rsync -av --delete Project joeuser@data.peregrine.hpc.uwm.edu:Data
```

The general syntax for pulling files from another host is:

```
shell-prompt: rsync -av --delete [username@]hostname:[source-path] destination-path
```

Example 7.3 Pulling data with rsync
The following command synchronizes the directory ~joeuser/Data/Project on Peregrine to ./Project on the local machine:

```
shell-prompt: rsync -av --delete joeuser@data.peregrine.hpc.uwm.edu:Data/project .
```

If you omit "username@" from the source or destination, rsync will try to log into the remote system with your username on the local system.

If you omit destination-path from a push command or source-path from a pull command, rsync will use your home directory on the remote host.

The command-line flags used above have the following meanings:

- **-a** Use archive mode. Archive mode copies all subdirectories recursively and preserves as many file attributes as possible, such as ownership, permissions, etc.

- **-v** Verbose copy: Display names of files and directories as they are copied.

- **--delete** Delete files and directories from the destination if they do not exists in the source. Without --delete, rsync will add and replace files in the destination, but never remove anything.

Caution
Note that a trailing "/" on source-path affects where rsync stores the files on the destination system. Without a trailing "/", rsync will create a directory called "source-path" under "destination-path" on the destination host. With a trailing "/" on source-path, destination-path is assumed to be the directory that will replace source-path on the destination host. This feature is a somewhat cryptic method of allowing you to change the name of the directory during the transfer. However, it is compatible with the basic Unix `cp` command.

Note also that the trailing "/" only affects the command when applied to source-path. A trailing "/" on destination-path has no effect.

The command below creates an identical copy of the directory Data/Model in Model (/home/bacon/Data/Model to be precise) on data.peregrine.hpc.uwm.edu. The resulting directory is the same regardless of whether the destination directory existed before the command or not.

```
shell-prompt: rsync -av --delete Model bacon@data.peregrine.hpc.uwm.edu:Data
```

The command below dumps the contents of Model directly into Data, and deletes everything else in the Data directory! In other words, it makes the destination directory Data identical to the source directory Model.

```
shell-prompt: rsync -av --delete Model/ bacon@data.peregrine.hpc.uwm.edu:Data
```

To achieve the same effect as the command with no "/", you would need to fully specify the destination path:

```
shell-prompt: rsync -av --delete Model/ bacon@data.peregrine.hpc.uwm.edu:Data/Model
```

Note that if using globbing on the remote system, any globbing patterns must be protected from expansion by the local shell by escaping them or enclosing them in hard quotes. We want the pattern expanded on the remote system, not the local system:
For full details on the rsync command, type:

```
shell-prompt: man rsync
```

### 7.14.2 File Transfer from Windows without Cygwin

If you're using Cygwin from Windows, you can utilize the rsync command as discussed in Section 7.14.1, provided you’ve installed the Cygwin rsync package. Otherwise, WinSCP provides a simple way to transfer files to and from your Windows PC. WinSCP is a free program that can be downloaded and installed in a few minutes from [http://winscp.net](http://winscp.net).

After installing WinSCP, simply launch the program, and the following dialog appears:

![The WinSCP login dialog](image)

WinSCP uses the secure shell protocol to connect to a remote system. Like the Unix `ssh` command, if this is the first time connecting from this computer, you will be asked if you want to add the host key and continue:
A WinSCP session

Once you’ve successfully logged in, you can simply drag files or directories from one system to the other. If you’re updating a large directory that already exists on the destination machine, you may want to check the “New and updated file(s) only” box. This will cause WinSCP to transfer only the files that are different on each end. This feature is a crude approximation to the functionality of rsync.

7.14.3 Self-test

1. Show the simplest Unix command to accomplish each of the following:
   
   (a) Copy or synchronize the directory ./PCB-Study to ~/PCB-Study under the user "joeuser" on the host data.peregrine.hpc.uwm.edu.
   
   (b) Copy or synchronize the directory ~/PCB-Study under the user "joeuser" on the host data.peregrine.hpc.uwm.edu to ./PCB-Study.

7.15 Environment Variables

Every Unix process maintains a list of character string variables called the environment. When a new process is created, it inherits the environment from the process that created it (its parent process).

Since the shell creates a new process whenever you run an external command, the shell’s environment can be used to pass information down to any command that you run. For example, text editors and other programs that manipulate the full terminal
screen need to know what type of terminal you are using. Different types of terminals use different magic sequences to move the
cursor, clear the screen, scroll, etc. To provide this information, we set the shell’s environment variable TERM to the terminal
type (usually "xterm"). When you run a command from the shell, it inherits the shell’s TERM variable, and therefore knows the
correct magic sequences for your terminal.

The **printenv** shows all of the environment variables currently set in your shell process.

```
shell-prompt: printenv
```

Setting environment variables requires a different syntax depending on which shell you are using. Most modern Unix shells are
extensions of either Bourne shell (sh) or C shell (csh), so there are only two variations of most shell commands that we need to
know for most purposes.

For Bourne shell derivatives, we use the **export** command:

```
shell-prompt: TERM=xterm
shell-prompt: export TERM
```

For C shell derivatives, we use **setenv**:

```
shell-prompt: setenv TERM xterm
```

The **PATH** variable specifies a list of directories containing external Unix commands. When you type a command at the shell
prompt, the shell checks the directories listed in **PATH** in order to find the command you typed. For example, when you type the
**ls** command, the shell utilizes **PATH** to locate the program in `/bin/ls`.

The directory names within in **PATH** are separated by colons. A simple value for **PATH** might be `/bin:/usr/bin:/usr/
local/bin`. When you type **ls**, the shell first checks for the existence of `/bin/ls`. If it does not exist, the shell then checks
for `/usr/bin/ls`, and so on, until it either finds the program or has checked all directories in **PATH**. If the program is not
found, the shell issues an error message such as "**ls**: Command not found".

Environment variables can be set from the shell prompt using the **export** command in Bourne shell and its derivatives (sh, bash,
ksh):

```
shell-prompt: export PATH='/bin:/usr/bin:/usr/local/bin'
```

or using **setenv** in C shell and its derivatives (csh, tcsh):

```
shell-prompt: setenv PATH '/bin:/usr/bin:/usr/local/bin'
```

### 7.15.1 Self-test

1. What are environment variables?
2. Does a Unix process have any environment variables when it starts? If so, where do they come from?
3. How can environment variables be used to communicate information to child processes?
4. Describe one common environment variable that it typically set by the shell and used by processes running under the shell.
5. Show how to set the environment variable TERM to the value "xterm" in
   
   (a) Bourne shell (sh)
   (b) Korn shell (ksh)
   (c) Bourne again shell (bash)
   (d) C shell (csh)
   (e) T-shell (tcsh)
7.16 Shell Variables

In addition to the environment, shells maintain a similar set of variables for their own use. These variables are not passed down to child processes, and are only used by the shell.

Shell variables can be arbitrary, but each shell also treats certain variable names specially. One common example is the shell variable that stores the shell prompt.

In Bourne-shell derivatives, this variable is called PS1. To set a shell variable in Bourne-shell derivatives, we use a simple assignment. (The export command above actually sets a shell variable called TERM and then exports it to the environment.)

```
shell-prompt: PS1="peregrine: 
```

In C shell derivatives, the variable is called prompt, and is set using the `set` command:

```
shell-prompt: set prompt="peregrine: 
```

Note The syntax for `set` is slightly different than for `setenv`. `Set` uses an `=` while `setenv` uses a space.

Shell prompt variables may contain certain special symbols that represent dynamic information they you might want to include in your shell prompt, such as the host name, command counter, current working directory, etc. Consult the documentation for your shell for details.

In all shells, you can view the current shell variables by typing `set` with no arguments:

```
shell-prompt: set
```

7.16.1 Self-test

1. Show how to set the shell prompt to "Peregrine: " in:
   (a) Bourne shell
   (b) C shell

2. How can you view a list of all current shell variables and their values?

7.17 More Shell Tools

7.17.1 Redirection and Pipes

Device Independence

Unix systems employ the concept of device independence. Unix device independence works by treating virtually every input and output device as a file. All input and output, whether to/from a file on a disk, a keyboard, a mouse, a scanner, or a printer, is simply a stream of bytes to be input or output by a program.

Most I/O devices are actually accessible as a device file in /dev. For example, the primary CD-ROM might be /dev/cd0, and the main disk might be /dev/ad0.

Data is often recovered from corrupted file systems or accidentally deleted files by reading the raw disk partition as a file using standard Unix commands such as grep!

```
shell-prompt: grep string /dev/ad0s1f
```

To see the raw input from a mouse as it is being moved, one could use the following command:

```
shell-prompt: hexdump /dev/mouse
```

`cat /dev/mouse` would also work, but the binary data stream would appear as garbage on the terminal screen.
Redirection

Since I/O devices and files are so interchangeable, Unix shells provide a facility called **redirection** to easily interchange them for any command without the command even knowing it.

Redirection depends on the notion of a **file stream**. You can think of a file stream as a hose connecting a program to a particular file or device. Redirection simply disconnects the hose from the default file or device and connects it to another one chosen by the shell user.

Every Unix process has three standard streams that are open from the moment the process is born. The standard streams are normally connected to the terminal, as shown in Table 7.8.

<table>
<thead>
<tr>
<th>Stream</th>
<th>Purpose</th>
<th>Default Connection</th>
</tr>
</thead>
<tbody>
<tr>
<td>Standard Input</td>
<td>User input</td>
<td>Terminal keyboard</td>
</tr>
<tr>
<td>Standard Output</td>
<td>Normal output</td>
<td>Terminal screen</td>
</tr>
<tr>
<td>Standard Error</td>
<td>Errors and warnings</td>
<td>Terminal screen</td>
</tr>
</tbody>
</table>

Table 7.8: Standard Streams

Redirection in the shell allows any or all of the three standard streams to be disconnected from the terminal and connected to a file or other I/O device. It uses operators within the commands to indicate which stream(s) to redirect and where. The basic redirection operators shells are shown in Table 7.9.

<table>
<thead>
<tr>
<th>Operator</th>
<th>Shells</th>
<th>Redirection type</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;</td>
<td>All</td>
<td>Standard Input</td>
</tr>
<tr>
<td>&gt;</td>
<td>All</td>
<td>Standard Output (overwrite)</td>
</tr>
<tr>
<td>&gt;&gt;</td>
<td>All</td>
<td>Standard Output (append)</td>
</tr>
<tr>
<td>2&gt;</td>
<td>Bourne-based</td>
<td>Standard Error (overwrite)</td>
</tr>
<tr>
<td>2&gt;&gt;</td>
<td>Bourne-based</td>
<td>Standard Error (append)</td>
</tr>
<tr>
<td>&gt;&amp;</td>
<td>C shell-based</td>
<td>Standard Output and Standard Error (overwrite)</td>
</tr>
<tr>
<td>&gt;&gt;&amp;</td>
<td>C shell-based</td>
<td>Standard Output and Standard Error (append)</td>
</tr>
</tbody>
</table>

Table 7.9: Redirection Operators

**Note** Memory trick: The arrow in a redirection operator points in the direction of data flow.

**Caution**

Using output redirection (>, 2>, or >>) in a command will normally overwrite (clobber) the file that you’re redirecting to, even if the command itself fails. Be very careful not to use output redirection accidentally. This most commonly occurs when a careless user meant to use input redirection, but pressed the wrong key. The moment you press Enter after typing a command containing "-> filename", filename will be erased! Remember that the shell performs redirection, not the command, so filename is clobbered before the command even begins running.

If *noclobber* is set for the shell, output redirection to a file that already exists will result in an error. The *noclobber* option can be overridden by appending a ! to the redirection operator in C shell derivatives or a | in Bourne shell derivatives. For example, >! can be used to force overwriting a file in csh or tcsh, and >! can be used in sh, ksh, or bash.

shell-prompt: ls > listing.txt  # Overwrite with listing of .
shell-prompt: ls /etc >> listing.txt  # Append listing of /etc
Note that redirection is performed by the shell, not the program. In the examples above, the `ls` command sends its output to the standard output. It is unaware that the standard output has been redirected to the file `listing.txt`.

Put another way, `listing.txt` is not an argument to the `ls` command. The redirection is handled by the shell, and `ls` runs as if it had been typed as simple:

```
shell-prompt: ls
```

More often than not, we want to redirect both normal output and error messages to the same place. This is why C shell and its derivatives use a combined operator that redirects both at once. The same effect can be achieved with Bourne-shell derivatives using another operator that redirects one stream to another stream. In particular, we redirect the standard output (stream 1) to a file (or device) and at the same time redirect the standard error (stream 2) to stream 1.

```
shell-prompt: find / -name '*.c' > list.txt 2>&1
```

If a program takes input from the standard input, we can redirect input from a file as follows:

```
shell-prompt: command < input-file
```

For example, consider the "bc" (binary calculator) command, an arbitrary-precision calculator which inputs numerical expressions from the standard input and writes the results to the standard output:

```
shell-prompt: bc
3.14159265359 * 4.2 ^ 2 + sqrt(30)
60.89491440932
quit
```

In the example above, the user entered "3.14159265359 * 4.2 ^ 2 + sqrt(30)" and "quit" and the bc program output "60.89491440932". We can place the input shown above in a file using any text editor, such as nano or vi, or by any other means:

```
shell-prompt: cat > bc-input.txt
3.14159265359 * 4.2 ^ 2 + sqrt(30)
quit
(Type Ctrl+d to signal the end of input to the cat command)
shell-prompt: more bc-input.txt
3.14159265359 * 4.2 ^ 2 + sqrt(30)
quit
```

Now that we have the input in a file, we can feed it to the bc command using input redirection instead of retyping it on the keyboard:

```
shell-prompt: bc < bc-input.txt
60.29203070318
```

### Special Files in /dev

Although it may seem a little confusing and circular, the standard streams themselves are represented as device files on Unix systems. This allows us to redirect one stream to another without modifying a program, by appending the stream to one of the device files `/dev/stdout` or `/dev/stderr`. For example, if a program sends output to the standard output and we want to send it instead to the standard error, we could do the following:

```
printf "Oops!" >> /dev/stderr
```

If we would like to discard output sent to the standard output or standard error, we can redirect it to `/dev/null`. For example, to see only error messages (standard error) from myprog, we could do the following:

```
./myprog > /dev/null
```

To see only normal output and not error messages, assuming Bourne shell:

```
./myprog 2> /dev/null
```

The device `/dev/zero` is a readable file that produces a stream of zero bytes.

The device `/dev/random` is a readable file that produces a stream of random integers in binary format.
**Pipes**

Quite often, we may want to use the output of one program as input to another. Such a thing could be done using redirection, as shown below:

```bash
shell-prompt: sort names.txt > sorted-names.txt
shell-prompt: uniq < sorted-names.txt > unique-names.txt
```

The same task can be accomplished in one command using a *pipe*. A pipe redirects one of the standard streams, just as redirection does, but to another process instead of to a file or device. In other words, we can use a pipe to send the standard output and/or standard error of one process directly to the standard input of another process.

**Example 7.4 Simple Pipe**

The command below uses a pipe to redirect the standard output of the `sort` command directly to the standard input of the `uniq`.

```bash
shell-prompt: sort names.txt | uniq > uniq-names.txt
```

Since a pipe runs multiple commands in the same shell, it is necessary to understand the concept of *foreground* and *background* processes, which are covered in detail in Section 7.18.

Multiple processes can output to a terminal at the same time, although the results would obviously be chaos in most cases. Only one process can receiving input from the keyboard, however.

The *foreground process* running under a given shell process is defined as the process that receives the input from the standard input device (usually the keyboard). This is the only difference between a foreground process and a background process.

When running a pipeline command, the last process in the pipeline is the foreground process. All others run in the background, i.e. do not use the standard input device inherited from the shell process. Hence, when we run:

```bash
shell-prompt: find /etc | more
```

It is the `more` command that receives input from the keyboard. The more command has its standard input redirected from the standard output of `find`, and the standard input of the `find` command is effectively disabled.

The more command is somewhat special: Since its standard input is redirected from the pipe, it opens another stream to connect to the keyboard so that the user can interact with it, pressing the space bar for another screen, etc.

For piping stderr, the notation is similar to that used for redirection:

<table>
<thead>
<tr>
<th>Operator</th>
<th>Shells</th>
<th>Pipe stream(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>`</td>
<td>`</td>
<td>All</td>
</tr>
<tr>
<td>`</td>
<td>&amp;`</td>
<td>C shell family</td>
</tr>
<tr>
<td>`2</td>
<td>`</td>
<td>Bourne shell family</td>
</tr>
</tbody>
</table>

Table 7.10: Pipe Operators

The entire chain of commands connected by pipes is known as a *pipeline*.

This is such a common practice that Unix has defined the term *filter* to apply to programs that can be used in this way. A filter is any command that can receive input from the standard input and send output to the standard output. Many Unix commands are designed to accept a file names as arguments, but also to use the standard input and/or standard output if no filename arguments are provided.

**Example 7.5 Filters**

The `more` command is commonly used as a filter. It can read a file whose name is provided as an argument, but will use the standard input if no argument is provided. Hence, the following two commands have the same effect:

```bash
shell-prompt: more names.txt
shell-prompt: more < names.txt
```
The only difference between these two commands is that in the first, the `more` receives `names.txt` as a command line argument, opens the file itself (creating a new file stream), and reads from the new stream (not the standard input stream). In the second instance, the shell opens the file and connects the standard input stream of the `more` command to it.

Using the filtering capability of `more`, we can paginate the output of any command:

```
shell-prompt: ls | more
shell-prompt: find . -name ‘*.c’ | more
shell-prompt: sort names.txt | more
```

We can string any number of commands together using pipes:

```
shell-prompt: ls | sort | uniq | more
```

One more useful tool worth mentioning is the `tee` command. The `tee` is a simple program that reads from its standard input and writes to both the standard output and to one or more files whose names are provided on the command line. This allows you to view the output of a program on the screen and redirect it to a file at the same time.

```
shell-prompt: ls | tee listing.txt
```

Recall that Bourne-shell derivatives do not have combined operators for redirecting standard output and standard error at the same time. Instead, we redirect the standard output to a file or device, and redirect the standard error to the standard output using `2>&1`.

We can use the same technique with a pipe, but there is one more condition: For technical reasons, the `2>&1` must come before the pipe.

```
shell-prompt: ls | tee listing.txt 2>&1 # Won’t work
shell-prompt: ls 2>&1 | tee listing.txt # Will work
```

The `yes` command produces a stream of y’s followed by newlines. It is meant to be piped into a program that prompts for y’s or n’s in response to yes/no questions, so that the program will receive a yes answer to all of its prompts and run without user input.

```
yes | ./myprog
```

### 7.17.2 Output Capture

Sometimes we want to use the output of a command not as an input stream, but as literal text, in place of a string constant or shell or environment variable.

We can achieve this by enclosing the command in back-quotes (``) or in `$()$`.

Print the output of the `date` command as part of a single line of text:

```
shell-prompt: printf "Today’s date is %s.\n" ‘date’
shell-prompt: printf "Today’s date is %s.\n" $(date)
```

Edit all the text files in the current working directory that contain the word "cheese":

```
shell-prompt: vi ‘fgrep -l cheese *.txt’
```

### 7.17.3 Self-test

1. What does device independence mean?

2. Show a Unix command that could be used to view the data stream sent by a mouse represented as `/dev/mse0`.

3. Name and describe the three standard streams available to all Unix processes.

4. Show the simplest Unix command to accomplish each of the following:
(a) Save a list of all files in /etc to the file list.txt.
(b) Compile prog.c under bash using gcc, saving error messages to errors.txt and normal screen output to output.txt.
(c) Compile prog.c under tcsh using gcc, saving both error messages and normal screen output to output.txt.
(d) Compile prog.c under tcsh using gcc, saving both error messages and normal screen output to output.txt. Overwrite output.txt even if noclobber is set.
(e) Run the program /prog1, causing it to use the file input.txt as the standard input instead of the keyboard.
(f) Compile prog.c under tcsh using gcc, saving both error messages and normal screen output to output.txt and sending them to the screen at the same time.

5. Which program in a pipeline runs as the foreground process?
6. How many programs can be included in a single pipeline?
7. What is a filter program?
8. Show a Unix command that will edit all the C program files in the subdirectory Programs, using the vi editor.

7.18 Process Control

Unix systems provide many tools for managing and monitoring processes that are already running.

Note that these tools apply to local Unix processes only. On distributed systems such as clusters and grids, job management is done using networked schedulers such as HTCondor, Grid Engine, or PBS.

It is possible to have multiple processes running under the same shell session. Such processes are considered either foreground processes or background processes. The foreground process is simply the process that receives the keyboard input. There can be no more than one foreground process under a given shell session, for obvious reasons.

Note that all processes, both foreground and background, can send output to the terminal at the same time, however. It is up to the user to ensure that output is managed properly and not intermixed.

There are three types of tools for process management, described in the following subsections.

7.18.1 External Commands

Unix systems provide a variety of external commands that monitor or manipulate processes based on their process ID (PID). A few of the most common commands are described below.

ps lists the currently running processes.

```
shell-prompt: ps [-a]  # BSD
shell-prompt: ps [-e]  # SYSV
```

ps is one of the rare commands whose options vary across different Unix systems. There are only two standards to which it may conform, however. The BSD version uses -a to indicate that all processes (not just your own) should be shown. System 5 (SYSV) ps uses -e for the same purpose. Run man ps on your system to determine which flags should be used.

kill sends a signal to a process (which may kill the process, but could serve other purposes).

```
shell-prompt: kill [-#] pid
```

The pid (process ID) is determined from the output of ps.

The signal number is an integer value following a -, such as -9. If not provided, the default signal sent is the TERM (terminate) signal.

Some processes ignore the TERM signal. Such processes can be force killed using the KILL (9) signal.
shell-prompt: kill -9 2342

Run `man signal` to learn about all the signals that can be issued with `kill`.

shell-prompt: ps

<table>
<thead>
<tr>
<th>PID</th>
<th>TT</th>
<th>STAT</th>
<th>TIME</th>
<th>COMMAND</th>
</tr>
</thead>
<tbody>
<tr>
<td>41167</td>
<td>0</td>
<td>Is</td>
<td>0:00.25</td>
<td>tcsh</td>
</tr>
<tr>
<td>78555</td>
<td>0</td>
<td>S+</td>
<td>0:01.98</td>
<td>ape unix.dbk</td>
</tr>
</tbody>
</table>

shell-prompt: kill 78555

The `killall` command will kill all processes running the program named as the argument. This eliminates the need to find the PID first, and is more convenient for killing multiple processes running the same program.

shell-prompt: killall fdtd

### 7.18.2 Special Key Combinations

Ctrl+c sends a terminate signal to the current foreground process. This usually kills the process immediately, although it is possible that some processes will ignore the signal.

Ctrl+z sends a suspend signal to the current foreground process. The process remains in memory, but does not execute further until it receives a resume signal (usually sent by running `fg`).

Ctrl+s suspends output to the terminal. This does not technically control the process directly, but has the effect of blocking any processes that are sending output, since they will stop running until the terminal begins accepting output again.

Ctrl+q resumes output to the terminal if it has been suspended.

### 7.18.3 Internal Shell Commands and Symbols

`jobs` lists the processes running under the current shell, but using the shell’s job IDs instead of the system’s process IDs.

Caution Shell jobs are ordinary processes running on the local system and should not be confused with cluster and grid jobs, which are managed by networked schedulers.

shell-prompt: jobs

`fg` brings a background job into the foreground.

shell-prompt: fg [%job-id]

There cannot be another job already running in the foreground. If no job ID is provided, and multiple background jobs are running, the shell will choose which background job to bring to the foreground. A job ID should always be provided if more than one background job is running.

`bg` resumes a job suspended by Ctrl+z in the background.

shell-prompt: prog
Ctrl+z
shell-prompt: bg
shell-prompt:

An & at the end of any command causes the command to be immediately placed in the background. It can be brought to the foreground using `fg` at any time.
nice runs a process at a lower than normal priority.

If (and only if) other processes in the system are competing for CPU time, they will get a bigger share than processes run under nice.

time runs a command under the scrutiny of the time command, which keeps track of the process’s resource usage.

There are both internal and external implementations of the time command. Run which time to determine which one your shell is configured to use.

nohup allows you to run a command that will continue after you log out. Naturally, all input and output must be redirected away from the terminal in order for this to work.

Bourne shell and compatible:

nohup ./myprogram < inputfile > outputfile 2>&1

C shell and compatible:

nohup ./myprogram < inputfile >& outputfile

This is often useful for long-running commands and where network connections are not reliable.

7.18.4 Self-test

1. What is a process?

2. What is the difference between a foreground process and a background process?

3. How many foreground processes can be running at once under a single shell process? Why?

4. How many background processes can be running at once under a single shell process? Why?

5. Show the simplest Unix command that will accomplish each of the following:
   (a) List all processes currently running.
   (b) List processes owned by you.
   (c) Kill the process with ID 7243.
   (d) Kill all processes running the program netsim.
   (e) Kill the process with ID 7243 after the first attempt failed.

6. How do you perform each of the following tasks?
   (a) Kill the current foreground process.
   (b) Suspend the current foreground process.
   (c) Resume a suspended process in the foreground.
   (d) Resume a suspended process in the background.
   (e) Start a new process, placing it in the background immediately.
   (f) Suspend terminal output for a process without suspending the process itself.
   (g) Resume suspended terminal output.
   (h) List the currently running jobs as seen by the shell.
   (i) Return job #2 to the foreground.
   (j) Run the program netsim at a reduced priority so that other processes will respond faster.
   (k) Run the program netsim and report the CPU time used when it finishes.
7.19 Running a Desktop Unix System

Most mainstream operating systems today are Unix compatible. Microsoft Windows is the only mainstream operating system that is not Unix compatible, but there are free compatibility systems available for Windows to provide some degree of compatibility and interoperability with Unix.

The de facto standard of Unix compatibility for Windows is Cygwin, http://cygwin.com, which is free and installs in about 10 minutes. There are alternatives to Cygwin, but Cygwin is the easiest to use and offers by far the most features and software packages.

Note None of the Unix compatibility systems for Windows are nearly as fast as a genuine Unix system on the same hardware, but they are fast enough for most purposes. If you want to maximize performance, there are many BSD Unix and Linux systems available for free.

Another option for running Unix programs on a Windows computer is to use a virtual machine (VM). This is discussed in Section 7.22.

Lastly, many Windows programs can be run directly under Unix, without a virtual machine running Windows, if the Unix system is running on x86-based hardware. This is accomplished using WINE, a Windows API emulator. WINE attempts to emulate the entire Windows system, as opposed to virtual machines, which emulate hardware. Emulating Windows is more ambitious, but eliminates the need to install and maintain a separate Windows operating system. Instead, the Windows applications run directly under Unix, with the WINE compatibility layer between them and the Unix system.

While it is possible to create a Unix-like environment under Windows using a system such as Cygwin, such systems have some inherent limitations in their capabilities and performance. Installing a Unix-compatible operating system directly has many benefits, especially for those developing their own code to run on the cluster.

Many professional quality Unix-based operating systems are available free of charge, and with no strings attached. These systems run a wide variety of high-quality free software, as well as many commercial applications. Hence, it is possible for researchers to develop Unix-compatible programs at very low cost that will run both on their personal workstation or laptop, and a cluster or grid.

Arguably the easiest Unix system to install and manage is TrueOS, a free, open source derivative of FreeBSD with a simple graphical installer, "Control Panel", and software manager:
A TrueOS system running KDE desktop.

TrueOS is extremely easy to install and manage, as well as extremely reliable. If you want to try out Unix while encountering as few hurdles as possible, TrueOS is probably your best bet.

TrueOS also allows you to install multiple desktop environments and choose among them each time you log in. It uses the ZFS file system to provide high performance, reliability and data protection.

Almost as easy to manage as TrueOS are the Ubuntu family of Linux systems (Ubuntu, Kubuntu, Xubuntu, Edubuntu, ...). Each of these Linux distributions is built on Debian Linux, with a different desktop environment. (Ubuntu uses Gnome, Kubuntu KDE, Xubuntu XFCE, etc.)

Back on the BSD side, there are also DesktopBSD and GhostBSD, both of which are derived from FreeBSD and have graphical installers and management tools.

The Debian system itself has also become relatively easy to install and manage in recent years. It lacks some of the bells and whistles of Ubuntu, but may be a bit faster and more stable as a result.

All of these systems have convenient methods for installing security updates and minor software upgrades.

When it comes time for a serious upgrade of the OS, don’t bother with upgrade tools. Back up your important files, reformat the disk, do a fresh install of the newer version, and restore your files.

Many hours are wasted trying to fix systems that have been broken by upgrades or were broken before the upgrade. It would have been faster and easier in many cases to run a backup and do a fresh install. You will need to do fresh installs sometimes anyway, so you might as well become good at it and use it as your primary method.

7.20 Software Management

7.20.1 The Stone Age vs. Today

There are many thousands of quality open source applications and libraries available for Unix systems.
Just knowing what exists can be a daunting task. Fortunately, software management systems such as the FreeBSD ports have organized documentation about what is available. You can browse software packages by category on the ports website: http://www.freebsd.org/ports/index.html. Even if you don’t use FreeBSD, this software listing is a great resource just for finding out what’s available.

Installing various open source packages can also be a daunting task, especially since the developers use many different programming languages and build systems.

Many valuable man-hours are lost to stone-age software management, i.e. manually downloading, unpacking, patching, building, and installing open source software.

Free software isn’t very free if it takes 20 hours of someone’s time to get it running. An average professional has a total cost to their employer on the order of $50/hour. Hence, 20 hours of their time = $1,000. If 1,000 professionals around the world spend an average of 20 hours installing the same software package, then $1,000,000 worth of highly valuable man-hours have gone to waste. Even worse, many people eventually give up on installing software entirely, so there are no gains to balance this loss.

In most cases, the software could have been installed in seconds using a software management system (SMS) and all that time could have been spent doing something productive.

When choosing a Unix system to run, a good ports or packages system is an important consideration. A ports or packages system automatically downloads and installs software from the Internet. Such systems also automatically install additional prerequisite ports or packages required by the package you requested.

For example, to install Firefox, you would first need to install dozens of libraries and other utilities that Firefox requires in order to run properly. (When you install Firefox on Windows or Max OS X, you are actually installing a bundle of all these packages.)

The ports or packages system will install all of them automatically when you choose to install Firefox. This allows you install software in seconds or minutes that might otherwise take days or weeks for an inexperienced programmer to manually download, patch, and compile.

### 7.20.2 Lots and Lots of Package Managers

Three operating systems, Debian Linux (and its derivatives such as Ubuntu), FreeBSD (and derivatives such as TrueOS), and Gentoo Linux, are far ahead of the rest in terms of software management. Each of them offers automated installation of more than 25,000 software packages.

A direct comparison of which collection is "biggest" is not really feasible. A raw count of Debian packages will produce a higher number than the others, but this is in part due to the Debian tradition of creating separate packages for documentation (-doc packages) and header files (-dev packages). FreeBSD ports and many other package managers traditionally include documentation and headers in the core package. For example, below are listings of FFTW (fast Fourier transform) packages on Debian and FreeBSD:

**Debian (documentation and header files in separate -doc and -dev packages):**

- fftw-dev
- fftw-docs
- fftw-double-dev
- fftw-single-dev
- fftw2
- fftw2-double
- fftw2-single
- fftw3
- fftw3-dev
- fftw3-doc

**FreeBSD (documentation and header files included with the core packages):**

- fftw
- fftw-float
- fftw3
- fftw3-float
fftw3-long
fftw3-quad

The bottom line, though, is that any of these three package managers is likely to provide most or all of the software that you will need.

By comparison, Red Hat Enterprise Linux (RHEL) and its derivatives such as CentOS and Scientific Linux, offer only a few thousand unique packages in the Yum repository. Although a few thousand may sound like a large number, most researchers will find that much of the software they need cannot be installed via Yum.

Software in the RHEL Yum repository also tends to be older versions than you will find in Debian, FreeBSD, or Gentoo packages. This is in no way a criticism of Red Hat Enterprise Linux, but simply illustrates that it was designed for a different purpose, namely highly stable enterprise servers running commercial software. The packages in Yum are intended to remain binary compatible with commercial applications for 7 years, not to provide the latest open source applications.

The Pkgsrc software management system can be used to maintain more recent open source software on enterprise Linux systems, separate from the commercial software and support software installed via Yum. Pkgsrc offers over 17,000 packages, most of which are tested on CentOS.

Pkgsrc has the additional benefit of being the only cross-platform SMS. It can currently be used to install most packages on NetBSD, Dragonfly BSD, Linux, Mac OS X, and Interix (a MS Windows Unix layer).

Even if you have to manually build and install your own software, you can probably install most of the the prerequisites from an SMS. Doing so will save you time and avoid conflicting installations of the same package on your system.

Better yet, learn how to create packages for an SMS, so that others don’t have to duplicate your effort in the future. Each SMS is a prime example of global collaboration, where each user leverages the work of thousands of others. The best way to support this collaboration is by contributing to it, even if in a very small way.

If you need to run commercial Linux applications, then an enterprise Linux system is the right choice. Many commercial applications are only supported on enterprise Linux systems such as RHEL and SUSE Enterprise. They often prove difficult to install and run reliably on other distributions.

For running mainly open source software, however, Debian, FreeBSD, or Gentoo will save you a great deal of time and effort.

On a FreeBSD system, installing Firefox is a matter of entering the following command:

shell-prompt: pkg install firefox

Similarly, on a Debian Linux system, you might use the following:

shell-prompt: apt-get install firefox

In either case, you just sit back for a minute or so while the package system downloads and installs everything you need to run Firefox.

7.20.3 Choosing Your Unix the Smart Way

Ultimately, the only thing that matters with respect to which Unix system you use is how well it runs the programs you need.

Many people make the mistake of choosing an operating system based on how "nice" it looks, what their friends (who often are not very computer savvy) are using, or how easy it is to install.

What’s really important, though, is what happens after the system is up and running. The effort required to maintain it over the course of a couple years is by far the lion’s share of the total cost of ownership, so get informed about what that cost will be for your particular needs before deciding.

Each operating system has its own focus, which may be very different from the rest.

The free operating systems include several systems based on BSD (Berkeley Systems Distribution), a free derivative of the original AT&T Unix from the University of California, Berkeley. It is the basis for FreeBSD, Mac OS X, NetBSD, OpenBSD, and a few others.
FreeBSD is the most popular among the free BSD-based systems. FreeBSD is known for its speed, ease of setup, robust network stack, and most of all its unparalleled stability. It is the primary server operating system used by Yahoo!, Netflix, and WhatsApp. Netflix alone accounted for more than 1/3 of streaming Internet traffic in North America in 2015. (See http://appleinsider.com/articles/16/01/20/netflix-boasts-37-share-of-internet-traffic-in-north-america-compared-with-3-for-apples-itunes) FreeBSD is also the basis of advanced file servers such as FreeNAS, Isilon, NAS4Free, NetApp, and Panasas, and network equipment from Juniper Networks, and the open source pfSense firewall.

The FreeBSD ports system makes it trivial to install any of more than 27,000 packages, including most mainstream scientific software. FreeBSD ports can be installed automatically either from a binary package, or from source code if you desire different build options than the packages provide.

Mac OS X is essentially FreeBSD with Apple’s proprietary user interface, so OS X users already have a complete Unix system on their Mac. In order to develop programs under Mac OS, you will need to install a compiler. Apple’s development system, called Xcode, is available as a free download. The free and open source MacPorts system offers the ability to easily install thousands of software packages directly from the Internet. The MacPorts system is one of the most modern and robust ports systems available for any operating system. There are also other package managers for Mac OS X, such as Fink, Homebrew, and Pkgsrc.

There are many free Linux distributions, as well as commercial versions such as Red Hat Enterprise and SUSE Enterprise. The most popular free distributions for personal use are currently Mint and the Ubuntu line (Ubuntu, Kubuntu, and Xubuntu), which are based on Debian Linux. These systems are known for their ease of installation and maintenance, and cutting-edge new Linux features. Systems based on Debian Linux support the Debian packages system, which offers more than 40,000 packages available for easy installation.

**Caution** Be careful about judging the popularity of different operating systems based on package counts. The Debian packages collection is somewhat inflated by the fact that they tend to split a single software distribution into several packages. For example, libraries are typically provided by at least three packages, a base package for only the runtime components (shared libraries), a -devel package for building your own programs with the library (header files), and a -doc package containing man/info pages, HTML, PDF, etc. Some libraries are further split into single/double precision libs, core and optional components, etc. Many other package systems, such as FreeBSD ports and pkgsrc, tend to provide all library components in a single package.

Gentoo Linux is a Linux system based heavily on ports. The Gentoo system installation process is very selective, and results in a compact, efficient system for each user’s particular needs. Gentoo is not as easy to install as other Linux systems, but is a great choice for more experienced Linux users who want to maximize performance and stability. Like FreeBSD and Debian, Gentoo’s ports system, known as portage, offers automated installation of nearly 20,000 software packages at the time of this writing.

The NetBSD project is committed to simplicity and portability. For this reason, NetBSD runs on far more hardware configurations than any other operating system.

The OpenBSD is run largely by computer security experts. Core security software such as OpenSSL and OpenSSH are developed by the OpenBSD project and used by most other operating systems.

Redhat Enterprise and SUSE Enterprise Linux are more conservative Linux-based systems designed to provide the stability required in enterprise environments. They are popular in corporate and academic data centers, where they support critical services, often running commercial software applications. They do not include the latest cutting edge Linux features, as doing so would jeopardize the stability they are meant to provide. They are based on older Linux kernels and system tools, which have been well-tested and debugged by users of cutting-edge Linux systems over several years.

### 7.20.4 Self-test

1. Which of today’s mainstream operating systems are Unix compatible and which are not?
2. Can MS Windows run Unix programs? If so, how?
3. Can non-Windows operating systems run Windows programs? If so, how?
7.21 Advanced: Enabling Remote Graphics

Most users will not need to run graphical applications on a remote Unix system. If you know that you will need to use a graphical user interface with your research software, or if you want to use a graphical editor such as gedit or emacs on over the network, read on. Otherwise, you can skip this section for now.

Unix uses a networked graphics interface called the X Window system. It is also sometimes called simply X11 for short. ( X11 is the latest major version of the system. ) X11 allows programs running on a Unix system to display graphics on the local screen or the screen of another Unix system on the network. The programs are called clients, and they display graphical output by sending commands to the X11 server on the machine where the output is to be displayed. Hence, your local computer must be running an X11 server in order to display Unix graphics, regardless of whether the client programs are running on your machine or another.

7.21.1 Configuration Steps Common to all Operating Systems

Modern Unix systems such as BSD, Linux, and Mac OS X have most of the necessary tools and configuration in place for running remote graphical applications.

However, some additional steps may be necessary on your computer to allow remote systems to access your display. This applies to all computers running an X11 server, regardless of operating system. Some additional steps that may be necessary for Cygwin systems are discussed in Section 7.21.2.

If you want to run graphical applications on a remote computer over an ssh connection, you will need to forward your local display to the remote system. This can be done for a single ssh session by providing the -X flag:

```
shell-prompt: ssh -X joe@login.peregrine.hpc.uwm.edu
```

This causes the ssh command to inform the remote system that X11 graphical output should be sent to your local display through the ssh connection. ( This is called SSH tunneling. )

---

**Caution** Allowing remote systems to display graphics on your computer can pose a security risk. For example, a remote user may be able to display a false login window on your computer in order to collect login and password information.

---

If you want to forward X11 connections to all remote hosts for all users on the local system, you can enable X11 forwarding in your ssh_config file (usually found in /etc or /etc/ssh) by adding the following line:

```ini
ForwardX11 yes
```

---

**Caution** Do this only if you are prepared to trust all users of your local system as well as all remote systems to which they might connect.

---

Some X11 programs require additional protocol features that can pose more security risks to the client system. If you get an error message containing "Invalid MIT-MAGIC-COOKIE" when trying to run a graphical application over an ssh connection, try using the -Y flag with ssh to open a trusted connection.

```
shell-prompt: ssh -Y joe@login.peregrine.hpc.uwm.edu
```

You can establish trusted connections to all hosts by adding the following to your ssh_config file:

```ini
ForwardX11Trusted yes
```
7.21.2 Graphical Programs on Windows with Cygwin

It is possible for Unix graphical applications on the remote Unix machine to display on a Windows machine, but this will require installing additional Cygwin packages and performing a few configuration steps on your computer in addition to those discussed in Section 7.21.1.

Installation

You will need to install the x11/xinit and x11/xhost packages using the Cygwin setup utility. This will install an X11 server on your Windows machine.

Configuration

After installing the Cygwin X packages, there are additional configuration steps:

1. Create a working ssh_config file by running the following command from a Cygwin shell window:

   ```shell
   shell-prompt: cp /etc/defaults/etc/ssh_config /etc
   ```

2. Then, using your favorite text editor, update the new /etc/ssh_config as described in Section 7.21.1.

3. Add the following line to .bashrc or .bash_profile (in your home directory):

   ```bash
   export DISPLAY=:0.0
   ```

   Cygwin uses bash for all users by default. If you are using a different shell, then edit the appropriate startup script instead of .bashrc or .bash_profile.

   This is not necessary when running commands from an xterm window (which is launched from Cygwin-X), but is necessary if you want to launch X11 applications from a Cygwin bash terminal which is part of the base Cygwin installation, and not X11-aware.

Start-up

To enable X11 applications to display on your Windows machine, you need to to start the X11 server on Windows by clicking Start → All Programs → Cygwin-X → XWin Server. The X server icon will appear in your Windows system tray to indicate that X11 is running. You can launch an xterm terminal emulator from the system tray icon, or use the Cygwin bash terminal, assuming that you have set your DISPLAY variable.

7.22 Advanced: Running Multiple Operating Systems

You don’t necessarily need to maintain a second computer in order to run Unix in addition to Windows. All mainstream Unix operating systems can be installed on a PC alongside Windows on a separate partition, or installed in a virtual machine (VM), such as Oracle VirtualBox, which is also available for free.

VMs are software packages that pretend to be computer hardware. You can install an entire operating system plus the software you need on the VM as if it were a real computer. The OS running under the VM is called the guest OS, and the OS running the VM on the real hardware is called the host.
Computational code runs at the same speed in the guest operating system as it does in the host. The main limitation imposed on guest operating systems is graphics speed. If you run applications requiring fast 3D rendering, such as video players, they should be run on the host operating system.

There are many VMs available for x86-based PC hardware, including VirtualBox, http://www.virtualbox.org/, which is free and open source, and runs on many different host platforms including FreeBSD, Linux, Mac OS X, Solaris, and Windows.

Running a Unix guest in a VM on Windows or Windows as a guest under Unix will provide a cleaner and more complete Unix experience than can be achieved with a compatibility layer like Cygwin. The main disadvantage of a VM is the additional disk space and memory required for running two operating systems at once. However, given the low cost of today’s hardware, this doesn’t usually present a problem on modern PCs.

Virtual machines are most often used to run Windows as a guest on a Unix system, to provide access to Windows-only applications to Unix (including Mac) users without maintaining a second computer. This configuration is best supported, and offers the most seamless integration between host and guest. An example is shown in Figure 7.2.

Another issue is that Windows systems need to be rebooted frequently, often several times per week, to activate security updates. Most Unix systems, on the other hand, can run uninterrupted for months at a time. (FreeBSD systems will typically run for years, if your power source is that stable.) There are far fewer security updates necessary for Unix systems, and most updates can be installed without rebooting. Rebooting a host OS requires rebooting all guests as well, but rebooting a guest OS does not affect the host. Hence, it’s best to run the most stable system as the host.

If necessary, it is possible to run Unix as a guest under Windows. FreeBSD and many Linux distributions are fully supported as VirtualBox guest operating systems. A VirtualBox disk image containing a FreeBSD installation is available at http://www.peregrine.hpc.uwm.edu. This image is provided in a standard format so that it can be easily imported into virtual machines such as VirtualBox and VMware.

7.23 Where to Learn More

There is a great deal of information available on the web. There are also many length books dedicated to Unix, which can provide more detail than this tutorial.

If you simply want to know what commands are available on your system, list the bin directories!
shell-prompt: ls /bin /usr/bin /usr/local/bin | more
Chapter 8

Unix Shell Scripting

Before You Begin
Before reading this chapter, you should be familiar with basic Unix concepts (Chapter 7) and the Unix shell (Section 7.4.3).

8.1 What is a Shell Script?

A shell script is essentially a file containing a sequence of Unix commands. A script is a type of program, but is distinguished from other programs in that it represents programming at a higher level.

While a typical program is largely made of calls to subprograms, a script contains invocations of whole programs.

In other words, a script is a way of automating the execution of multiple separate programs in sequence.

The Unix command-line structure was designed to be convenient for both interactive use and for programming in scripts. Running a Unix command is much like calling a subprogram. The difference is just syntax. A subprogram call in C encloses the arguments in parenthesis and separates them with commas:

function_name(arg1, arg2, arg3);

A Unix command is basically the same, except that it uses spaces instead of parenthesis and commas:

command_name arg1 arg2 arg3

8.1.1 Self-test

1. What is a shell script?

2. How are Unix commands similar to and different from subprogram calls in a language like C?

8.2 Scripts vs Programs

It is important to understand the difference between a "script" and a "real program", and which languages are appropriate for each.

Scripts tend to be small (no more than a few hundred or a few thousand lines of code) and do not do any significant computation of their own.

Instead, scripts run "real programs" to do most of the computational work. The job of the script is simply to automate and document the process of running programs.
As a result, scripting languages do not need to be efficient and are generally interpreted rather than compiled. (Interpreted language programs run an order of magnitude or more slower than equivalent compiled programs, unless most of their computation is done by built-in, compiled subprograms.)

Real programs may be quite large and may implement complex computational algorithms. Hence, they need to be fast and as a result are usually written in compiled languages.

If you plan to use exclusively pre-existing programs such as Unix commands and/or add-on application software, and need only automate the execution of these programs, then you need to write a script and should choose a good scripting language.

If you plan to implement your own algorithm(s) that may require a lot of computation, then you need to write a program and should select an appropriate compiled programming language.

8.2.1 Self-test

1. How do scripts differ from programs written in languages like C or Fortran?
2. Why would it not be a good idea to write a matrix multiplication program as a Bourne shell script?

8.3 Why Write Shell Scripts?

8.3.1 Efficiency and Accuracy

Any experienced computer user knows that we often end up running basically the same sequence of commands many times over. Typing the same sequence of commands over and over is a waste of time and highly prone to errors.

All Unix shells share a feature that can help us avoid this repetitive work: They don’t care where their input comes from.

It is often said that the Unix shell reads commands from the keyboard and executes them. This is not really true. The shell reads commands from any input source and executes them. The keyboard is just one common input source that can be used by the shell. Ordinary files are also very commonly used as shell input.

Recall from Chapter 7 that Unix systems employ device independence, which means that any Unix program that reads from a keyboard can also read the same input from a file or any other input device.

Hence, if we’re going to run the same sequence of commands more than once, we don’t need to retype the sequence each time. The shell can read the commands from anywhere, not just from the keyboard. We can put those commands into a text file once and tell the shell to read the commands from the file, which is much easier than typing them all again.

Rule of Thumb If you might have to do it again, script it.

In theory, Unix commands could also be piped in from another program or read from any other device attached to a Unix system, although in practice, they usually come from the keyboard or a script file.

Self-test

1. Describe two reasons for writing shell scripts.
2. Are there Unix commands that you can run interactively, but not from a shell script? Explain.
3. What feature of Unix makes shell scripts so convenient to implement?
4. What is a good rule of thumb for deciding whether to write a shell script?
8.3.2 Documentation

There is another very good reason for writing shell scripts in addition to saving us a lot of redundant typing:

*A shell script is the ultimate documentation of the work we have done on a computer.*

By writing a shell script, we record the exact sequence of commands needed to reproduce results, in perfect detail. Hence, the script serves a dual purpose of automating and documenting our processes.

Developing a script has a ratchet effect on your knowledge. Once you add a command to a script, you will never forget how to use it for that task.

**Rule of Thumb** A Unix user should never find themselves trying to remember how they did something. Script it the first time...

Clear documentation of our work flow is important in order to justify research funding and to be able to reproduce results months or years later.

Imagine that we instead decided to run our sequence of commands manually and document what we did in a word processor. First, we’d be typing everything twice: Once at the shell prompt and again into the document.

The process of typing the same commands each time would be painful enough, but to document it in detail while we do it would be distracting. We’d also have to remember to update the document every time we type a command differently. This is hard to do when we’re trying to focus on getting results.

Writing a shell script allows us to stay focused on perfecting the process. Once the script is finished and working perfectly, we have the process perfectly documented. We can and should add comments to the script to make it more readable, but even without comments, the script itself preserves the process in detail.

Many experienced users will never run a processing command from the keyboard. Instead, they only put commands into a script and run and re-run the script until it’s finished.

**Self-test**

1. Describe another good reason for writing shell scripts.
2. Why is it so important to document the sequence of commands used?

8.3.3 Why Unix Shell Scripts?

There are many scripting languages to choose from, including those used on Unix systems, like Bourne shell, C shell, Perl, Python, etc., as well as some languages confined to other platforms like Visual Basic (Microsoft Windows only) and AppleScript (Apple only).

Note that the Unix-based scripting languages can be used on any platform, including Microsoft Windows (with Cygwin, for example) and Apple’s Mac OS X, which is Unix-compatible out of the box.

Once you learn to write Unix shell scripts, you’re prepared to do scripting on any computer, without having to learn another language.

**Self-test**

1. What are two advantages of writing Unix shell scripts instead of using a scripting language such as Visual Basic or AppleScript?

8.3.4 Self-test

1. Describe three reasons for writing shell scripts instead of running commands from the keyboard.
8.4 Which Shell?

8.4.1 Common Shells

When writing a shell script, there are essentially two scripting languages to choose from: Bourne shell and C shell. These were the first two popular shells for Unix, and all common shells that have come since are compatible with one or the other.

The most popular new shells are Bourne Again shell (bash), which is an extension of Bourne shell, Korn shell (ksh), which is another extension of Bourne shell, Z-shell, a very sophisticated extension of Bourne shell, and T-shell (TENEX C shell, tcsh), which is an extended C shell.

- Bourne shell family
  - Bourne shell (sh)
  - Bourne-again shell (bash)
  - Korn shell (ksh)
  - Z-shell (zsh)

- C shell family
  - C shell (csh)
  - T-shell (tcsh)

Both Bourne shell and C shell have their own pros and cons. C shell syntax is cleaner, more intuitive, and more similar to the C programming language (hence the name C shell). However, C shell lacks some features such as subprograms (although C shell scripts can run other C shell scripts, which is arguably a better approach in many situations).

Bourne shell is used almost universally for Unix system scripts, while C shell is fairly popular in scientific research.

Note Every Unix system has a Bourne shell in /bin/sh. Hence, using vanilla Bourne shell (not bash, ksh, or zsh) for scripts maximizes their portability by ensuring that they will run on any Unix system without the need to install any additional shells.

8.4.2 Self-test

1. What is one advantage of Bourne shell over C shell?
2. What is one advantage of C shell over Bourne shell?

8.5 Writing and Running Shell Scripts

A shell script is a simple text file and can be written using any Unix text editor. Some discussion of Unix text editors can be found in Section 7.11.4.

Caution Recall from Section 7.10.1 that Windows uses a slightly different text file format than Unix. Hence, editing Unix shell scripts in a Windows editor can be problematic. Users are advised to do all of their editing on a Unix machine rather than write programs and scripts on Windows and transfer them to Unix.
Shell scripts often contain very complex commands that are wider than a typical terminal window. A command can be continued on the next line by typing a backslash (\) immediately before pressing Enter. This feature is present in all Unix shells. Of course, it can be used on an interactive CLI as well, but is far more commonly used in scripts to improve readability.

```bash
printf "This command is too long to fit in a single 80-column terminal window, so we break it up with a backslash."
```

It’s not a bad idea to name the script with a filename extension that matches the shell it uses. This just makes it easier to see which shell each of your script files use. Table 8.1 shows conventional filename extensions for the most common shells. However, if a script is to be installed into the PATH so that it can be used as a regular command, it is usually given a name with no extension. Most users would rather type "cleanup" than "cleanup.bash".

Like all programs, shell scripts should contain comments to explain what the commands in it are doing. In all Unix shells, anything from a ‘#’ character to the end of a line is considered a comment and ignored by the shell.

```bash
# Print the name of the host running this script
hostname
```

<table>
<thead>
<tr>
<th>Shell</th>
<th>Extension</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bourne Shell</td>
<td>.sh</td>
</tr>
<tr>
<td>C shell</td>
<td>.csh</td>
</tr>
<tr>
<td>Bourne Again Shell</td>
<td>.bash</td>
</tr>
<tr>
<td>T-shell</td>
<td>.tcsh</td>
</tr>
<tr>
<td>Korn Shell</td>
<td>.ksh</td>
</tr>
<tr>
<td>Z-shell</td>
<td>.zsh</td>
</tr>
</tbody>
</table>

Table 8.1: Conventional script filename extensions

**Practice Break**

Using your favorite text editor, enter the following text into a file called `hello.sh`.

1. The first step is to create the file containing your script, using any text editor, such as nano:

   ```bash
   shell-prompt: nano hello.sh
   ```

   Once in the text editor, add the following text to the file:

   ```bash
   printf "Hello!\n"
   printf "I am a script running on a computer called 'hostname'\n"
   ```

   After typing the above text into the script, save the file and exit the editor. If you are using nano, the menu at the bottom of the screen tells you how to save (write out, Ctrl+o) and exit (Ctrl+x).

2. Once we’ve written a script, we need a way to run it. A shell script is simply input to a shell program. Like many Unix programs, shells take their input from the standard input by default. We could, therefore, use redirection to make it read the file via standard input:

   ```bash
   shell-prompt: sh < hello.sh
   ```

Sync-point: Instructor: Make sure everyone in class succeeds at this exercise before moving on.

Since the shell normally reads commands from the standard input, the above command will "trick" sh into reading its commands from the file `hello.sh`.

However, Unix shells and other scripting languages provide a more convenient method of indicating what program should interpret them. If we add a special comment, called a *shebang line* to the top of the script file and make the file executable using
**chmod**, the script can be executed like a Unix command. We can then simply type its name at the shell prompt, and another shell process will start up and run the commands in the script. If the directory containing such a script is included in $PATH, then the script can be run from any current working directory, just like `ls`, `cp`, etc.

The shebang line consists of the string "#!" followed by the full path name of the command that should be used to execute the script, or the path /usr/bin/env followed by the name of the command. For example, both of the following are valid ways to indicate a Bourne shell (sh) script, since /bin/sh is the Bourne shell command.

```
#!/bin/sh
```

```
#!/usr/bin/env sh
```

When you run a script as a command, by simply typing its filename at the Unix command-line, a new shell process is created to interpret the commands in the script. The shebang line specifies which program is invoked for the new shell process that runs the script.

**Note** The shebang line must begin at the very first character of the script file. There cannot even be blank lines above it or white space to the left of it. The "#!" is an example of a **magic number**. Many files begin with a 16-bit (2-character) code to indicate the type of the file. The "#!" indicates that the file contains some sort of interpreted language program, and the characters that follow will indicate where to find the interpreter.

The /usr/bin/env method is used for add-on shells and other interpreters, such as Bourne-again shell (bash), Korn shell (ksh), and Perl (perl). These interpreters may be installed in different directories on different Unix systems. For example, bash is typically found in /bin/bash on Linux systems, /usr/local/bin/bash on FreeBSD systems, and /usr/pkg/bin/bash on NetBSD. The T-shell is found in /bin/tcsh on FreeBSD and CentOS Linux and in /usr/bin/tcsh on Ubuntu Linux.

The env command is found in /usr/bin/env on virtually all Unix systems. Hence, this provides a method for writing shell scripts that are portable across Unix systems (i.e. they don’t need to be modified to run on different Unix systems).

**Note**

Every script or program should be tested on more than one platform (e.g. BSD, Cygwin, Linux, Mac OS X, etc.) immediately, in order to shake out bugs before they cause problems.

The fact that a program works fine on one operating system and CPU does not mean that it’s free of bugs. By testing it on other operating systems, other hardware types, and with other compilers or interpreters, you will usually expose bugs that will seem obvious in hindsight.

As a result, the software will be more likely to work properly when time is critical, such as when there is an imminent deadline approaching and no time to start over from the beginning after fixing bugs. Encountering software bugs at times like these is very stressful and usually easily avoided by testing the code on multiple platforms in advance.

Some shells, such as Bourne shell (sh) and C shell (csh) are virtually always installed in /bin, so their shebang lines normally specify the full path name (#!/bin/sh or #!/bin/csh).

```
#!/bin/sh
```

```
#!/bin/csh
```

```
#!/bin/bash  (Bad idea: Not portable)
```

```
#!/bin/tcsh  (Bad idea: Not portable)
```

```
#!/usr/bin/env bash  (This is portable)
```

```
#!/usr/bin/env tcsh  (This is portable)
```

Using `/usr/bin/env` can pose a slight security risk, since it could potentially find and execute a Trojan horse. Hence, it is only used for interpreters that don’t have a standard location.

**Example 8.1 A Simple Bash Script**

Suppose we want to write a script that is always executed by bash, the Bourne Again Shell. We simply need to add a shebang line indicating the path name of the `bash` executable file.

```
shell-prompt: nano hello.bash
```

Enter the following text in the editor. Then save the file and exit back to the shell prompt.

```
#!/usr/bin/env bash
# A simple command in a shell script
printf "Hello, world!\n"
```

Now, make the file executable and run it:

```
shell-prompt: chmod a+rx hello.bash  # Make the script executable
shell-prompt: ./hello.bash  # Run the script as a command
```

**Example 8.2 A Simple T-shell Script**

Similarly, we might want to write a script that is always executed by tcsh, the TENEX C Shell. We simply need to add a shebang line indicating the path name of the `tcsh` executable file.

```
shell-prompt: nano hello.tcsh
```

Enter the following text in the editor. Then save the file and exit back to the shell prompt.

```
#!/usr/bin/env tcsh
# A simple command in a shell script
printf "Hello, world!\n"
```

Now, make the file executable and run it:

```
shell-prompt: chmod a+rx hello.tcsh  # Make the script executable
shell-prompt: ./hello.tcsh  # Run the script as a command
```

**Note** Many of the Unix commands you use regularly may actually be scripts rather than binary programs.

**Note**

There may be cases where you cannot make a script executable. For example, you may not own it, or the filesystem may not allow executables, to prevent users from running programs where they shouldn’t.

In these cases, we can simply run the script as an argument to an appropriate shell. For example:

```
shell-prompt: sh hello.sh
shell-prompt: bash hello.bash
shell-prompt: csh hello.csh
shell-prompt: tsh hello.tcsh
shell-prompt: ksh hello.ksh
```

Note also that the shebang line in a script is ignored when you explicitly run a shell this way. The content of the script will be interpreted by the shell that you have manually invoked, regardless of what the shebang line says.

Scripts that you create and intend to use regularly can be placed in your PATH, so that you can run them from anywhere. A common practice among Unix users is to create a directory called `~/bin`, and configure the login environment to that this directory is always in the PATH. Programs and scripts placed in this directory can then be used like any other Unix command, without typing the full path name.
8.5.1 Self-test

1. What tools can be used to write shell scripts?
2. Is it a good idea to write Unix shell scripts under Windows? Why or why not?
3. After creating a new shell script, what must be done in order to make it executable like a Unix command?
4. What is a shebang line?
5. What does the shebang line look like for a Bourne shell script? A Bourne again shell script? Explain the differences.

8.6 Shell Start-up Scripts

Each time you log into a Unix machine or start a new shell (e.g., when you open a new terminal), the shell process runs one or more special scripts called start up scripts. Some common start up scripts:

<table>
<thead>
<tr>
<th>Script</th>
<th>Shells that use it</th>
<th>Executed by</th>
</tr>
</thead>
<tbody>
<tr>
<td>/etc/profile, ~/.profile</td>
<td>Bourne shell family</td>
<td>Login shells only</td>
</tr>
<tr>
<td>File named by $ENV (typically .shrc or .shinit)</td>
<td>Bourne shell family</td>
<td>All interactive shells (login and non-login)</td>
</tr>
<tr>
<td>~/.bashrc</td>
<td>Bourne-again shell only</td>
<td>All interactive shells (login and non-login)</td>
</tr>
<tr>
<td>~/.bash_profile</td>
<td>Bourne-again shell only</td>
<td>Login shells only</td>
</tr>
<tr>
<td>~/.kshrc</td>
<td>Korn shell</td>
<td>All interactive shells (login and non-login)</td>
</tr>
<tr>
<td>/etc/csh.login, ~/.login</td>
<td>C shell family</td>
<td>Login shells only</td>
</tr>
<tr>
<td>/etc/csh.cshrc, ~/.cshrc</td>
<td>C shell family</td>
<td>All shell processes</td>
</tr>
<tr>
<td>~/.tcshrc</td>
<td>T-shell</td>
<td>All shell processes</td>
</tr>
</tbody>
</table>

Table 8.2: Shell Start Up Scripts

Note
Non-interactive Bourne-shell family shell processes, such as those used to execute shell scripts, do not execute any start up scripts. Hence, Bourne shell family scripts are not affected by start up scripts. In contrast, all C shell script processes execute ~/.cshrc if it exists. Hence, C shell family scripts are affected by ~/.cshrc. You can override this in C-shell scripts by invoking the shell with -f as follows:

```csh
#!/bin/csh -f
```

The man page for your shell has all the details about which start up scripts are invoked and when.

Start up scripts are used to configure your PATH and other environment variables, set your shell prompt and other shell features, create aliases, and anything else you want done when you start a new shell.

One of the most common alterations users make to their start up script is editing their PATH to include a directory containing their own programs and scripts. Typically, this directory is named ~/bin, but you can name it anything you want.

To set up your own ~/bin to store your own scripts and programs, do the following:

1. shell-prompt: mkdir ~/bin
2. Edit your start up script and add ~/bin to the PATH.
   If you’re using Bourne shell or Bourne-again shell, you can add ~/bin to your PATH for login shells only by adding the following to your .profile or .bash_profile:
If you’re using C shell or T-shell, add the following to your `.login`:

```
setenv PATH ${PATH} : ~/bin
```

---

**Caution**

Adding `~`/bin before (left of) `${PATH}` will cause your shell to look in `~`/bin before looking in the standard directories such as `/bin` and `/usr/bin`. Hence, if a binary or script in `~`/bin has the same name as another command, the one in `~`/bin will override it. This is considered a security risk, since users could be tricked into running a Trojan-horse `ls` or other common command if care is not taken to protect `~`/bin from modification. Hence, adding to the tail (right side) of PATH is usually recommended, especially for inexperienced users.

---

3. Update the PATH in your current shell process by sourcing the start up script, or by logging out and logging back in.

There is no limit to what your start up scripts can do, so you can use your imagination freely and find ways to make your Unix shell environment easier and more powerful.

### 8.6.1 Self-test

1. What is the purpose of a shell start-up script?
2. What are the limitations on what a start-up script can do compared to a normal script?

### 8.7 Sourcing Scripts

In some circumstances, we might not want a script to be executed by a separate shell process. For example, suppose we just made some changes to our `.cshrc` or `.bashrc` file that would affect PATH or some other important environment variable.

If we run the start up script by typing `~`/.cshrc or `~`/.bashrc, a new shell process will be started which will execute the commands in the script and then terminate. The shell you are using, which is the parent process, will be unaffected, since parent processes do not inherit environment from their children.

In order to make the "current" shell process run the commands in a script, we must `source` it. This is done using the internal shell command `source` in all shells except Bourne shell, which uses ".". Most Bourne-derived shells support both "." and "source".

Hence, to source `.cshrc`, we would run

```
shell-prompt: source ~/.cshrc
```

To source `.bashrc`, we would run

```
shell-prompt: source ~/.bashrc
```

or

```
. ~/.bashrc
```

To source `.shrc` from a basic Bourne shell, we would have to run

```
. ~/.shrc
```
8.7.1 Self-test

1. What is sourcing?
2. When would we want to source a script?

8.8 Scripting Constructs

Although Unix shells make no distinction between commands entered from the keyboard and those input from a script, there are certain shell features that are meant for scripting and not convenient or useful to use interactively.

Many of these features will be familiar to anyone who has done any computer programming. They include constructs such as comments, conditionals and loops.

The following sections provide a very brief introduction to shell constructs that are used in scripting, but generally not used on the command line.

8.9 Strings

A string constant in a shell script is anything enclosed in single quotes (‘this is a string’) or double quotes (“this is also a string”).

Unlike most programming languages, text in a shell scripts that is not enclosed in quotes and does not begin with a ‘$’ or other special character is also interpreted as a string constant. Hence, all of the following are the same:

shell-prompt: ls /etc
shell-prompt: ls "/etc"
shell-prompt: ls '/etc'

However, something contains white space (spaces or tabs), then it will be seen as multiple separate strings. The last example below will not work properly, since 'Program' and 'Files' are seen as separate arguments:

shell-prompt: cd 'Program Files'
shell-prompt: cd "Program Files"
shell-prompt: cd Program Files

Note
Special sequences such as ‘\n’ must be enclosed in quotes or escaped, otherwise the ‘\’ is seen as escaping the ‘n’.

Hello
 != "Hello\n"
"Hello\n" = ‘Hello\n’ = Hello\n
8.10 Output

Output commands are only occasionally useful at the interactive command line. We may sometimes use them to take a quick look at a variable such as $PATH.

shell-prompt: echo $PATH

Output commands are far more useful in scripts, and are used in the same ways as output statements in any programming language.

The echo command is commonly used to output something to the terminal screen:
However, `echo` should be avoided, since it is not portable across different shells and even the same shell on different Unix systems. There are many different implementations of `echo` commands, some internal to the shell and some external programs. Different implementations of `echo` use different command-line flags and special characters to control output formatting.

In addition, the output formatting capabilities of `echo` commands are extremely limited.

The `printf` command supersedes `echo`. It has a rich set of capabilities and is specified in the POSIX.2 standard, so its behavior is the same on all Unix systems.

The `printf` command is an external command, so it is independent of which shell you are using.

The functionality of `printf` closely matches that of the `printf()` function in the standard C library. It recognizes special characters such as `\n` (line feed), `\t` (tab), `\r` (carriage return), etc. and can print numbers in different bases.

The basic syntax of a `printf` command is as follows:

```
printf format-string argument [argument ...]
```

The format-string contains literal text and a `format specifier` to match each of the arguments that follows.

Each format specifier begins with a ` `%` and is followed by a symbol indicating the format in which to print the argument.

<table>
<thead>
<tr>
<th>Specifier</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>%s</td>
<td>String</td>
</tr>
<tr>
<td>%d</td>
<td>Decimal number</td>
</tr>
<tr>
<td>%o</td>
<td>Octal number</td>
</tr>
</tbody>
</table>

Table 8.3: Printf Format Specifiers

The `printf` command also recognizes most of the same special character sequences as the C `printf()` function:

<table>
<thead>
<tr>
<th>Sequence</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>\n</code></td>
<td>Newline (move down to next line)</td>
</tr>
<tr>
<td><code>\r</code></td>
<td>Carriage Return (go to beginning of current line)</td>
</tr>
<tr>
<td><code>\t</code></td>
<td>Tab (go to next tab stop)</td>
</tr>
</tbody>
</table>

Table 8.4: Special Character Sequences

```
printf "%s %d %o\n" 10 10 10
```

Output:

```
10 10 12
```

There are many other format specifiers and special character sequences. For complete information, run `man printf`.

To direct `printf` output to the standard error instead of the standard output, we simply take advantage of device independence and use redirection:

```
printf ‘Hello!\n’ >> /dev/stderr
```
Practice Break
Sync-point: Instructor: Make sure everyone in class succeeds at this exercise before moving on.
Write a shell script containing the printf statement above and run it. Write the same script using two different shells, such as Bourne shell and C shell. What is the difference between the two scripts?

8.10.1 Self-test

1. What are the advantages of printf over the echo command?

2. Does printf work under all shells? Why or why not?

8.11 Shell and Environment Variables

Variables are essential to any programming language, and scripting languages are no exception. Variables are useful for user input, control structures, and for giving short names to commonly used values such as long path names.

Most programming languages distinguish between variables and constants, but in shell scripting, we use variables for both.

Shell processes have access to two separate sets of string variables.

Recall from Section 7.15 that every Unix process has a set of string variables called the environment, which are handed down from the parent process in order to communicate important information.

For example, the TERM variable, which identifies the type of terminal a user is using, is used by programs such as top, vi, nano, more, and other programs that need to manipulate the terminal screen (move the cursor, highlight certain characters, etc.) The TERM environment variable is usually set by the shell process so that all of the shell’s child processes (those running vi, nano, etc.) will inherit the variable.

Unix shells also keep another set of variables that are not part of the environment. These variables are used only for the shell’s purpose and are not handed down to child processes.

There are some special shell variables such as "prompt" and "PS1" (which control the appearance of the shell prompt in C shell and Bourne shell, respectively).

Most shell variables, however, are defined by the user for use in scripts, just like variables in any other programming language.

8.11.1 Assignment Statements

In all Bourne Shell derivatives, a shell variable is created or modified using the same simple syntax:

```
varname=value
```

**Caution** In Bourne shell and its derivatives, there can be no space around the '='. If there were, the shell would think that 'varname' is a command, and '=' and 'value' are arguments.

```
bash-4.2$ name = Fred
bash: name: command not found
bash-4.2$ name=Fred
bash-4.2$ printf "$name
"
Fred
```

When assigning a string that contains white space, it must be enclosed in quotes or the white space characters must be escaped:
#!/usr/bin/env bash
name=Joe Sixpack   # Error
name="Joe Sixpack" # OK
name=Joe\ Sixpack  # OK

C shell and T-shell use the `set` command for assigning variables.

#!/bin/csh
set name="Joe Sixpack"

---

**Caution** Note that Bourne family shells also have a `set` command, but it has a completely different meaning, so take care not to get confused. The Bourne `set` command is used to set shell command-line options, not variables.

---

Unlike some languages, shell variables need not be declared before they are assigned a value. Declaring variables is unnecessary, since there is only one data type in shell scripts.

All variables in a shell script are character strings. There are no integers, Booleans, enumerated types, or floating point variables, although there are some facilities for interpreting shell variables as integers, assuming they contain only digits.

If you must manipulate real numbers in a shell script, you could accomplish it by piping an expression through `bc`, the Unix arbitrary-precision calculator:

```
printf "scale=5\n243.9 * $variable\n" | bc
```

Such facilities are very inefficient compared to other languages, however, partly because shell languages are interpreted, not compiled, and partly because they must convert each string to a number, perform arithmetic, and convert the results back to a string. Shell scripts are meant to automate sequences of Unix commands and other programs, not perform numerical computations.

In Bourne shell family shells, environment variables are set by first setting a shell variable of the same name and then `exporting` it.

```
TERM=xterm
export TERM
```

Modern Bourne shell derivatives such as bash (Bourne Again Shell) can do it in one line:

```
export TERM=xterm
```

**Note** Exporting a shell variable permanently tags it as exported. Any future changes to the variable’s value will automatically be copied to the environment. This type of linkage between two objects is very rare in programming languages: Usually, modifying one object has no effect on any other.

---

C shell derivatives use the `setenv` command to set environment variables:

```
setenv TERM xterm
```

---

**Caution** Note that unlike the `set` command, `setenv` requires white space, not an `='; between the variable name and the value.
8.11.2 Variable References

To reference a shell variable or an environment variable in a shell script, we must precede its name with a `'$'`. The `'$'` tells the shell that the following text is to be interpreted as a variable name rather than a string constant. The variable reference is then **expanded**, i.e. replaced by the value of the variable. This occurs anywhere in a command except inside a string bounded by single quotes or following an escape character (\), as explained in Section 8.9.

These rules are basically the same for all Unix shells.

```
#!/usr/bin/env bash
name="Joe Sixpack"
printf "Hello, name!\n"  # Not a variable reference!
printf "Hello, $name!\n"  # References variable "name"
```

Output:

```none
Hello, name!
Hello, Joe Sixpack!
```

**Practice Break**

Type in and run the following scripts:

```
#!/bin/sh
first_name="Bob"
last_name="Newhart"
printf "%s %s is the man.\n" $first_name $last_name
```

C SH version:

```
#!/bin/csh
set first_name="Bob"
set last_name="Newhart"
printf "%s %s is the man.\n" $first_name $last_name
```

**Note**

If both a shell variable and an environment variable with the same name exist, a normal variable reference will expand the shell variable.

In Bourne shell derivatives, a shell variable and environment variable of the same name always have the same value, since exporting is the only way to set an environment variable. Hence, it doesn’t really matter which one we reference.

In C shell derivatives, a shell variable and environment variable of the same name can have different values. If you want to reference the environment variable rather than the shell variable, you can use the `printenv` command:

```
Darwin heron bacon ~ 319: set name=Sue
Darwin heron bacon ~ 320: setenv name Bob
Darwin heron bacon ~ 321: echo $name
Sue
Darwin heron bacon ~ 322: printenv name
Bob
```

There are some special C shell variables that are automatically linked to environment counterparts. For example, the shell variable `path` is always the same as the environment variable `PATH`. The C shell man page is the ultimate source for a list of these variables.

If a variable reference is immediately followed by a character that could be part of a variable name, we could have a problem:
#!/usr/bin/env bash
name="Joe Sixpack"
printf "Hello to all the $names of the world!"

Instead of printing "Hello to all the Joe Sixpacks of the world", the printf will fail because there is no variable called "names". In Bourne Shell derivatives, non-existent variables are treated as empty strings, so this script will print "Hello to all the of the world!". C shell will complain that the variable "names" does not exist.

We can correct this by delimiting the variable name in curly braces:

#!/usr/bin/env bash
name="Joe Sixpack"
printf "Hello to all the ${name}s of the world!"

This syntax works for all shells.

### 8.11.3 Using Variables for Code Quality

Another very good use for shell variables is in eliminating redundant string constants from a script:

```
#!/usr/bin/env bash

output_value='myprog'
printf "$output_value
" >> Run2/Output/results.txt
more Run2/Output/results.txt
cp Run2/Output/results.txt latest-results.txt
```

If for any reason the relative path `Run2/Output/results.txt` should change, then you’ll have to search through the script and make sure that all instances are updated. This is a tedious and error-prone process, which can be avoided by using a variable:

```
#!/usr/bin/env bash

output_value='myprog'
output_file="Run2/Output/results.txt"
printf "$output_value
" >> $output_file
more $output_file
cp $output_file latest-results.txt
```

In the second version of the script, if the path name of `results.txt` changes, then only one change must be made to the script. Avoiding redundancy is one of the primary goals of any good programmer.

In a more general programming language such as C or Fortran, this role would be served by a constant, not a variable. However, shells do not support constants, so we use a variable for this.

In most shells, a variable can be marked read-only in an assignment to prevent accidental subsequent changes. Bourne family shells use the `readonly` command for this, while C shell family shells use `set -r`.

```
#!/bin/sh

readonly output_value='myprog'
printf "$output_value
" >> Run2/Output/results.txt
more Run2/Output/results.txt
cp Run2/Output/results.txt latest-results.txt
```

```
#!/bin/csh

set -r output_value='myprog'
printf "$output_value
" >> Run2/Output/results.txt
more Run2/Output/results.txt
cp Run2/Output/results.txt latest-results.txt
```
8.11.4 Output Capture

Output from a command can be captured and used as a string in the shell environment by enclosing the command in back-quotes (``). In Bourne-compatible shells, we can use $() in place of back-quotes.

```bash
#!/bin/sh -e

# Using output capture in a command
printf "Today is %s.\n" 'date'
printf "Today is %s.\n" $(date)

# Using a variable. If using the output more than once, this will
# avoid running the command multiple times.
today='date'
printf "Today is %s\n" $today
```

8.11.5 Self-test

1. Describe two purposes for shell variables.
2. Are any shell variable names reserved? If so, describe two examples.
3. Show how to assign the value "Roger" to the variable "first_name" in both Bourne shell and C shell.
4. Why can there be no spaces around the '=' in a Bourne shell variable assignment?
5. How can you avoid problems when assigning values that contain white space?
6. Do shell variables need to be declared before they are used? Why or why not?
7. Show how to assign the value "xterm" to the environment variable TERM in both Bourne shell and C shell.
8. Why do we need to precede variables names with a '$' when referencing them?
9. How can we output a letter immediately after a variable reference (no spaces between them). For example, show a printf statement that prints the contents of the variable fruit immediately followed by the letter 's'.

```bash
fruit=apple
# Show a printf that will produce the output "I have 10 apples", using
# the variable fruit.

```

10. How can variables be used to enhance code quality? From what kinds of errors does this protect you?
11. How can a variable be made read-only in Bourne shell? In C shell?

8.12 Hard and Soft Quotes

Double quotes are known as soft quotes, since shell variable references and some other special symbols are expanded when they are enclosed in soft quotes.

Single quotes are known as hard quotes, since every character inside single quotes is taken literally as part of the string. Nothing inside hard quotes is processed by the shell.

What will each of the following print? (If you’re not sure, try it!)

```bash
#!/usr/bin/env bash

name='Joe Sixpack'
printf "Hi, my name is $name.\n"
```
#!/usr/bin/env bash
name='Joe Sixpack'
printf 'Hi, my name is $name.
'

#滨州/bin/env bash
first_name='Joe'
last_name='Sixpack'
name='$first_name $last_name'
printf "Hi, my name is $name.
"

If you need to include a quote character as part of a string, you have two choices:

1. "Escape" it (precede it with a backslash character):
   
   ```bash
   printf 'Hi, I\'m Joe Sixpack.
   '
   ```

2. Use the other kind of quotes to enclose the string. A string terminated by double quotes can contain a single quote and vice-versa:

   ```bash
   printf "Hi, I\'m Joe Sixpack."
   printf 'You can use a " in here."
   ```

No special operators are needed to concatenate strings in a shell script. We can simply place multiple strings in any form (variable references, literal text, etc.) next to each other.

```bash
printf 'Hello,\$var.'   # Variable between two hard-quotes strings
printf "Hello, \$var."
# Variable between text in a soft-quoted string
```

### 8.12.1 Self-test

1. What is the difference between soft and hard quotes?

2. Show the output of the following script:

   ```bash
   #!/bin/sh
   name="Bill Murray"
   printf "$name
   printf "$name
   printf $name
   printf "$name
   ```

### 8.13 User Input

In Bourne Shell derivatives, data can be input from the standard input using the `read` command:

```bash
#!/usr/bin/env bash
printf "Please enter your name: "
read name
printf "Hello, $name!\n"
```

C shell and T-shell use a symbol rather than a command to read input:
#!/bin/csh
printf "Please enter your name: 
set name="$<"
printf "Hello, $name!\n"

The $< symbol behaves like a variable, which makes it more flexible than the read command used by Bourne family shells. It can be used anywhere a regular variable can appear.

#!/bin/csh
printf "Enter your name: 
printf "Hi, $<!\n"

Note The $< symbol should always be enclosed in soft quotes in case the user enters text containing white space.

Practice Break
Write a shell script that asks the user to enter their first name and last name, stores each in a separate shell variable, and outputs "Hello, first-name last-name".

Please enter your first name: Barney
Please enter your last name: Miller
Hello, Barney Miller!

8.13.1 Self-test

1. Do the practice break in this section if you haven’t already.

8.14 Conditional Execution

Sometimes we need to run a particular command or sequence of commands only if a certain condition is true. For example, if program B processes the output of program A, we probably won’t want to run B at all unless A finished successfully.

8.14.1 Command Exit Status

Conditional execution in Unix shell scripts often utilizes the exit status of the most recent command. All Unix programs return an exit status. It is not possible to write a Unix program that does not return an exit status. Even if the programmer neglects to explicitly return a value, the program will return a default value.

By convention, programs return an exit status of 0 if they determine that they completed their task successfully and a variety of non-zero error codes if they failed. There are some standard error codes defined in the C header file sysexits.h. You can learn about them by running man sysexits.

We can check the exit status of the most recent command by examining the shell variable $? in Bourne shell family shells or $status in C shell family shells.

bash> ls
myprog.c
bash> echo $?
0
bash> ls -z
ls: illegal option -- z
usage: ls [-ABCFGHILPRSTUWZabcdfghiklmnopqrstuvwxyz] [-D format] [file ...]
bash> echo $?
1
bash>

tcsh> ls
myprog.c
tcsh> echo $status
0
tcsh> ls -z
ls: illegal option -- z
usage: ls [-ABCFGHILPRSTUWZabcdfghiklmnopqrstuvwxyz] [-D format] [file ...]
tcsh> echo $status
1
tcsh>

Practice Break
Run several commands correctly and incorrectly and check the $? or $status variable after each one.

8.14.2 If-then-else Statements

All Unix shells have an if-then-else construct implemented as internal commands. The Bourne shell family of shells all use the same basic syntax. The C shell family of shells also use a common syntax, which is somewhat different from the Bourne shell family.

Bourne Shell Family

The general syntax of a Bourne shell family if statement is shown below. Note that there can be an unlimited number of elifs, but we will use only one for this example.

```bash
#!/bin/sh
if command1
then
  command
  command
  ...
elif command2
then
  command
  command
  ...
else
  command
  command
  ...
fi
```

Note
The 'if' and the 'then' are actually two separate commands, so they must either be on separate lines as shown above, or separated by an operator such as ';', which can be used instead of a newline to separate Unix commands.

cd; ls
if command; then
The if command executes command1 and checks the exit status when it completes.

If the exit status of command1 is 0 (indicating success), then all the commands before the elif are executed, and everything after the elif is skipped.

If the exit status is non-zero, then nothing above the elif is executed. The elif command then executes command2 and checks its exit status.

If the exit status of command2 is 0, then the commands between the elif and the else are executed and everything after the else is skipped.

If the exit status of command2 is non-zero, then everything above the else is skipped and everything between the else and the fi is executed.

**Note** In Bourne shell if statements, an exit status of zero effectively means 'true' and non-zero means 'false', which is the opposite of C and similar languages.

In most programming languages, we use some sort of Boolean expression (usually a comparison, also known as a relation), not a command, as the condition for an if statement.

This is generally true in Bourne shell scripts as well, but the capability is provided in an interesting way. We’ll illustrate by showing an example and then explaining how it works.

Suppose we have a shell variable and we want to check whether it contains the string "blue". We could use the following if statement to test:

```bash
#!/bin/sh
printf "Enter the name of a color: 
read color
if [ "$color" = "blue" ]; then
    printf "You entered blue.\n"
elif [ "$color" = "red" ]; then
    printf "You entered red.\n"
else
    printf "You did not enter blue or red.\n"
fi
```

The interesting thing about this code is that the square brackets are *not* Bourne shell syntax. As stated above, the Bourne shell if statement simply executes a command and checks the exit status. This is *always* the case.

The ']' in the condition above is actually an external command! In fact, is simply another name for the `test` command. The files `/bin/test` and `/bin/[` are actually the same program file:

```
FreeBSD tocino bacon ~ 401: ls -l /bin/test /bin/[
-r-xr-xr-x 2 root wheel 8516 Apr 9 2012 /bin/[
```

We could have also written the following:

```bash
if test "$color" = "blue"; then
```

Hence, "$color", =, "blue", and ] are all separate arguments to the [ command, and must be separated by white space. If the command is invoked as '[', then the last argument must be ']'. If invoked as 'test', then the ']' is not allowed.

The test command can be used to perform comparisons (relational operations) on variables and constants, as well as a wide variety of tests on files. For comparisons, test takes three arguments: the first and third are string values and the second is a relational operator.
# Compare a variable to a string constant
```
test "$name" = 'Bob'
```

# Compare the output of a program directly to a string constant
```
test 'myprog' = 42
```

For file tests, test takes two arguments: The first is a flag indicating which test to perform and the second is the path name of the file or directory.

# See if output file exists and is readable to the user
# running test
```
test -r output.txt
```

The exit status of `test` is 0 (success) if the test is deemed to be true and a non-zero value if it is false.

```
shell-prompt: test 1 = 1
shell-prompt: echo $?
0
shell-prompt: test 1 = 2
shell-prompt: echo $?
1
```

The relational operators supported by `test` are shown in Table 8.5.

<table>
<thead>
<tr>
<th>Operator</th>
<th>Relation</th>
</tr>
</thead>
<tbody>
<tr>
<td>=</td>
<td>Lexical equality (string comparison)</td>
</tr>
<tr>
<td>-eq</td>
<td>Integer equality</td>
</tr>
<tr>
<td>!=</td>
<td>Lexical inequality (string comparison)</td>
</tr>
<tr>
<td>-ne</td>
<td>Integer inequality</td>
</tr>
<tr>
<td>&lt;</td>
<td>Lexical less-than (10 &lt; 9)</td>
</tr>
<tr>
<td>-lt</td>
<td>Integer less-than (9 -lt 10)</td>
</tr>
<tr>
<td>-le</td>
<td>Integer less-than or equal</td>
</tr>
<tr>
<td>&gt;</td>
<td>Lexical greater-than</td>
</tr>
<tr>
<td>-gt</td>
<td>Integer greater-than</td>
</tr>
<tr>
<td>-ge</td>
<td>Integer greater-than or equal</td>
</tr>
</tbody>
</table>

Table 8.5: Test Relational Operators

**Caution**
Note that some operators, such as `<` and `>`, have special meaning to the shell, so they must be escaped or quoted.

```
test 10 > 9  # Redirects output to a file called '9'.
    # The only argument sent to the test command is '10'.
    # The test command issues a usage message since it requires
    # more arguments.
```

```
test 10 ">" 9  # Compares 10 to 9.
test 10 ">" 9  # Compares 10 to 9.
```

**Caution** It is a common error to use `==` with the test command, but the correct comparison operator is `=`.

Common file tests are shown in Table 8.6. To learn about additional file tests, run "man test".
<table>
<thead>
<tr>
<th>Flag</th>
<th>Test</th>
</tr>
</thead>
<tbody>
<tr>
<td>-e</td>
<td>Exists</td>
</tr>
<tr>
<td>-r</td>
<td>Is readable</td>
</tr>
<tr>
<td>-w</td>
<td>Is writable</td>
</tr>
<tr>
<td>-x</td>
<td>Is executable</td>
</tr>
<tr>
<td>-d</td>
<td>Is a directory</td>
</tr>
<tr>
<td>-f</td>
<td>Is a regular file</td>
</tr>
<tr>
<td>-L</td>
<td>Is a symbolic link</td>
</tr>
<tr>
<td>-s</td>
<td>Exists and is not empty</td>
</tr>
<tr>
<td>-z</td>
<td>Exists and is empty</td>
</tr>
</tbody>
</table>

Table 8.6:

**Caution**

Variable references in a [ or test command should usually be enclosed in soft quotes. If the value of the variable contains white space, such as "navy blue", and the variable is not enclosed in quotes, then "navy" and "blue" will be considered two separate arguments to the [ command, and the [ command will fail. When [ sees "navy" as the first argument, it expects to see a relational operator as the second argument, but instead finds "blue", which is invalid.

Furthermore, if there is a chance that a variable used in a comparison is empty, then we must attach a common string to the arguments on both sides of the operator. It can be almost any character, but '0' is popular and easy to read.

```bash
name=""
if [ "$name" = "Bob" ]; then   # Error, expands to: if [ = Bob; then
if [ 0"$name" = 0"Bob" ]; then # OK, expands to: if [ 0 = 0Bob ]; then
```

Relational operators are provided by the test command, not by the shell. Hence, to find out the details, we would run "man test" or "man []", not "man sh" or "man bash".

See Section 8.14.1 for information about using the test command.
Practice Break
Run the following commands in sequence and run `echo $?` after every test or `[` command under bash and `echo $status` after every test or `[` command under tcsh.

**bash**
```
test 1 = 1
test 1=2
test 1 = 2
[ 1 = 1
[ 1 = 1 ]
[ 1 = ’1’ ]
[1=1]
[ 2 < 10 ]
[ 2 \< 10 ]
[ 2 ’<’ 10 ]
[ 2 -lt 10 ]
[ $name = ’Bill’ ]
[ 0$name = 0’Bill’ ]
name=’Bob’
[ $name = ’Bill’ ]
[ $name = Bill ]
[ $name = Bob ]
exit
tcsh
[ $name = ’Bill’ ]
[ 0$name = 0’Bill’ ]
set name=’Bob’
[ $name = ’Bill’ ]
exit
```

**C shell Family**

Unlike the Bourne shell family of shells, the C shell family implements its own operators, so there is generally no need for the `test` or `[` command (although you can use it in C shell scripts if you really want to).

The C shell if statement requires () around the condition, and the condition is always a Boolean expression, just like in C and similar languages. As in C, and unlike Bourne shell, a value of zero is considered false and non-zero is true.

```
#!/bin/csh -ef

printf "Enter the name of a color: 
set color = "$<"
if ( "$color" == "blue" ) then
    printf "You entered blue.
else if ( "$color" == "red" ) then
    printf "You entered red.
else
    printf "You did not enter blue or red.
endif
```

The C shell relational operators are shown in Table 8.7.

**Note** C shell does not directly support string comparisons except for equality and inequality. To see if a string is lexically less-than or greater than another, use the test or `[` command with `<`, `<=`, `>`, or `>=`. 
## Operator Relation

<table>
<thead>
<tr>
<th>Operator</th>
<th>Relation</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;</td>
<td>Integer less-than</td>
</tr>
<tr>
<td>&gt;</td>
<td>Integer greater-than</td>
</tr>
<tr>
<td>&lt;=</td>
<td>Integer less-than or equal</td>
</tr>
<tr>
<td>&gt;=</td>
<td>Integer greater-than or equal</td>
</tr>
<tr>
<td>==</td>
<td>String equality</td>
</tr>
<tr>
<td>!=</td>
<td>String inequality</td>
</tr>
<tr>
<td>=~</td>
<td>String matches glob pattern</td>
</tr>
<tr>
<td>!~</td>
<td>String does not match glob pattern</td>
</tr>
</tbody>
</table>

Table 8.7: C Shell Relational Operators

Conditions in a C shell if statement do not have to be relations, however. We can check the exit status of a command in a C shell if statement using the `{}` operator:

```bash
#!/bin/csh
if ( { command } ) then
    command ...
endif
```

The `{}` essentially inverts the exit status of the command. If command returns 0, the value of `{ command }` is 1, which means "true" to the if statement. If command returns a non-zero status, then the value of `{ command }` is zero.

C shell if statements also need soft quotes around strings that contain white space. However, unlike the test command, it can handle empty strings, so we don’t need to add an arbitrary prefix like '0' if the string may be empty.

```bash
if [ 0"$name" = 0"Bob" ]; then
if ( "$name" == "Bob" ) then
    printf 'Hi, Bob!
' endif
set first_name=Bob
if ( $first_name == Bob ) then
    printf 'Hi, Bob!
' endif
exit
```

### Practice Break

Type in the following commands in sequence:

```bash
tcsh
if ( $first_name == Bob ) then
    printf 'Hi, Bob!
’
endif
set first_name=Bob
if ( $first_name == Bob ) then
    printf 'Hi, Bob!
’
endif
exit
```

### 8.14.3 Conditional Operators

The shell’s conditional operators allow us to alter the exit status of a command or utilize the exit status of each in a sequence of commands. They include the Boolean operators AND (&&), OR (||), and NOT (!).

```bash
shell-prompt: command1 && command2
shell-prompt: command1 || command2
shell-prompt: ! command
```
Table 8.8: Shell Conditional Operators

<table>
<thead>
<tr>
<th>Operator</th>
<th>Meaning</th>
<th>Exit status</th>
</tr>
</thead>
<tbody>
<tr>
<td>! command</td>
<td>NOT</td>
<td>0 if command failed, 1 if it succeeded</td>
</tr>
<tr>
<td>command1 &amp;&amp; command2</td>
<td>AND</td>
<td>0 if both commands succeeded</td>
</tr>
<tr>
<td>command1</td>
<td></td>
<td>command2</td>
</tr>
</tbody>
</table>

Note that in the case of the && operator, command2 will not be executed if command 1 fails (exits with non-zero status), since it could not change the exit status. Once any command in an && sequence fails, the exit status of the whole sequence will be 1, so no more commands will be executed.

Likewise in the case of a || operator, once any command succeeds (exits with zero status), the remaining commands will not be executed.

This fact is often used to conditionally execute a command only if another command is successful:

```bash
pre-processing && main-processing && post-processing
```

When using the `test` or `[` commands, multiple tests can be performed using either the shell’s conditional operators or the `test` command’s Boolean operators:

```bash
if [ 0"$first_name" = 0"Bob" ] && [ 0"$last_name" = 0"Newhart" ]; then
    if test 0"$first_name" = 0"Bob" && test 0"$last_name" = 0"Newhart"; then
        if [ 0"$first_name" = 0"Bob" -a 0"$last_name" = 0"Newhart" ]; then
            if test 0"$first_name" = 0"Bob" -a 0"$last_name" = 0"Newhart"; then
                The latter is probably more efficient, since it only executes a single `[` command, but efficiency in shell scripts is basically a lost cause, so it’s best to aim for readability instead. If you want speed, use a compiled language.

Conditional operators can also be used in a C shell if statement. Parenthesis are recommended around each relation for readability.

```bash
if ( ($first_name == "Bob") && ($last_name == "Newhart") ) then
```
Practice Break
Run the following commands in sequence and run `echo $?` after every command under bash and `echo $status` after every command under tcsh.

### bash

```bash
ls -z
ls -z && echo Done
ls -a && echo Done
ls -z || echo Done
ls -a || echo Done
first_name=Bob
last_name=Newhart
if [ 0"$first_name" = 0"Bob" ] && [ 0"$last_name" = 0"Newhart" ]
then
    printf 'Hi, Bob!
'fi
if [ 0"$first_name" = 0"Bob" -a 0"$last_name" = 0"Newhart" ]
then
    printf 'Hi, Bob!
'fi
exit
tcsh
ls -z
ls -z && echo Done
ls -a && echo Done
ls -z || echo Done
ls -a || echo Done
if ( $first_name == Bob && $last_name == Newhart ) then
    printf 'Hi, Bob!
'endif
set first_name=Bob
set last_name=Nelson
if ( $first_name == Bob && $last_name == Newhart ) then
    printf 'Hi, Bob!
'endif
set last_name=Newhart
if ( $first_name == Bob && $last_name == Newhart ) then
    printf 'Hi, Bob!
'endif
exit
```

### 8.14.4 Case and Switch Statements

If you need to compare a single variable to many different values, you could use a long string of elifs or else ifs:

```bash
#!/bin/sh
printf "Enter a color name: 
read color
if [ "$color" = "red" ] || 
    [ "$color" = "orange" ]; then
    printf "Long wavelength\n"
elif [ "$color" = "yellow" ] || 
    [ "$color" = "green" ] || 
    [ "$color" = "blue" ]; then
    printf "Medium wavelength\n"
elif [ "$color" = "indigo" ] || 
    [ "$color" = "violet" ]; then
```
Like most languages, however, Unix shells offer a cleaner solution.

**Bourne shell has the case statement:**

```sh
#!/bin/sh

printf "Enter a color name: 
read color

case $color in
   red|orange)
      printf "Long wavelength\n"
   ;;
   yellow|green|blue)
      printf "Medium wavelength\n"
   ;;
   indigo|violet)
      printf "Short wavelength\n"
   ;;
   *)
      printf "Invalid color name: $color\n"
   ;;
esacl
```

**C shell has a switch statement that looks almost exactly like the switch statement in C, C++, and Java:**

```csh
#!/bin/csh -ef

printf "Enter a color name: 
set color = "$<"

switch($color)
 case   red:
 case   orange:
    printf "Long wavelength\n"
 breaksw
 case   yellow:
 case   green:
 case   blue:
    printf "Medium wavelength\n"
 breaksw
 case   indigo:
 case   violet:
    printf "Short wavelength\n"
 breaksw
 default:
    printf "Invalid color name: $color\n"
endsw
```

**Note** The `;;` and `breaksw` statements cause a jump to the first statement after the entire case or switch. The `;;` is required after every value in the case statement. The `breaksw` is optional in the switch statement. If omitted, the script will simply continue on and execute the statements for the next case value.
8.14.5 Self-test

1. What is an "exit status"? What conventions to Unix programs follow regarding the exit status?

2. How can we find out the exit status of the previous command in Bourne shell? In C shell?

3. Write a Bourne shell script that uses an if statement to run 'ls -l > output.txt' and view the output using 'more' only of the ls command succeeded.

4. Repeat the previous problem using C shell.

5. Write a Bourne shell script that inputs a first name and outputs a different message depending on whether the name is 'Bob'.

```
shell-prompt: ./script
What is your name? Bob
Hey, Bob!
shell-prompt: ./script
What is your name? Bill
Hey, you're not Bob!
```

6. Repeat the previous problem using C shell.

7. Write a Bourne and/or C shell script that inputs a person’s age and indicates whether they get free peanuts. Peanuts are provided to senior citizens.

```
shell-prompt: ./script
How old are you? 42
Sorry, no peanuts for you.
shell-prompt: ./script
How old are you? 72
Have some free peanuts, wise sir!
```

8. Why is it necessary to separate the tokens between [ and ] with white space? What will happen if we don’t?

9. What will happen if a value being compared using test or [ contains white space? How to we remedy this?

10. What will happen if a value being compared using test or [ is an empty string? How to we remedy this?

11. Why do the < and > operators need to be escaped (\<, \>) or quoted when used with the test command?

12. What is the == operator used for with the test command?

13. How do we check the exit status of a command in a C shell if statement?

14. Write a Bourne and/or C shell script that inputs a person’s age and indicates whether they get free peanuts. Peanuts are provided to senior citizens and children between the ages of 3 and 12.

```
shell-prompt: ./script
How old are you? 42
Sorry, no peanuts for you.
shell-prompt: ./script
How old are you? 72
Have some free peanuts, wise sir!
```

15. Write a Bourne shell script that uses conditional operators to run 'ls -l > output.txt' and view the output using 'more' only of the ls command succeeded.

16. Write a shell script that checks the output of `uname` using a case or switch statement and reports whether the operating system is supported. Assume supported operating systems include Cygwin, Darwin, FreeBSD, and Linux.

```
shell-prompt: ./script
FreeBSD is supported.
```

```
shell-prompt: ./script
AIX is not supported.
```
8.15 Loops

We often need to run the same command or commands on a group of files or other data.

8.15.1 For and Foreach

Unix shells offer a type of loop that takes an enumerated list of string values, rather than counting through a sequence of numbers. This makes it more flexible for working with sets of files or arbitrary sets of values.

This type of loop is well suited for use with globbing (filename patterns using wild cards, as discussed in Section 7.8.5):

```bash
#!/usr/bin/env bash
# Process input-1.txt, input-2.txt, etc.
for file in input-*.txt
do
  ./myprog $file
done
```

```csh
#!/bin/csh -ef
# Process input-1.txt, input-2.txt, etc.
foreach file (input-*.txt)
do
  ./myprog $file
done
```

These loops are not limited to using filenames, however. We can use them to iterate through any list of string values:

```bash
#!/bin/sh
for fish in flounder gobie hammerhead manta moray sculpin
do
  printf "%s\n" $fish
done
```

```bash
#!/usr/bin/env bash
for c in 1 2 3 4 5 6 7 8 9 10
do
  printf "%d\n" $c
done
```

**Practice Break**

Type in and run the fish example above.

**Note** Note again that the Unix commands, including the shell, don’t generally care whether their input comes from a file or a device such as the keyboard. Try running the fish example by typing it directly at the shell prompt as well as by writing a script file. When running it directly, be sure to use the correct shell syntax for the interactive shell you are running.
8.15.2 While Loops

A for or foreach loop is not convenient for iterating through a long number sequence.

The while loop is a more general loop that iterates as long as some condition is true. It uses the same types of expressions as an if statement.

The while loop is often used to iterate through long integer sequences:

```bash
#!/usr/bin/env bash
c=1
while [ $c -le 100 ]
  do
    printf "%d\n" $c
    c=$(($c + 1))  # (( )) encloses an integer expression
  done
```

Note again that the [ above is an external command, as discussed in Section 8.14.1.

```bash
#!/bin/csh -ef
set c = 1
while ( $c <= 100 )
  printf "%d\n" $c
  @ c = $c + 1  # @ is like set, but indicates an integer expression
end
```

Practice Break
Type in and run the script above.

While loops can also be used to iterate until an input condition is met:

```bash
#!/bin/sh
continue=''
while [ 0"$continue" != 0’y’ ] && [ 0"$continue" != 0’n’ ]; do
  printf "Would you like to continue? (y/n) "
  read continue
  printf "$continue" "
done
```

```bash
#!/bin/csh -ef
set continue=''
while ( ("$continue" != ’y’) && ("$continue" != ’n’) )
  printf "Continue? (y/n) "
  set continue="$<"
end
```

Practice Break
Type in and run the script above.

We may even want a loop to iterate forever. This is often useful when using a computer to collect data at regular intervals:
#!/bin/sh

# 'true' is an external command that always returns an exit status of 0
while true; do
    sample-data # Read instrument
    sleep 10 # Pause for 10 seconds without using any CPU time
done

#!/bin/csh -ef

while ( 1 )
    sample-data # Read instrument
    sleep 10 # Pause for 10 seconds without using any CPU time
end

8.15.3 Self-test

1. Write a shell script that prints the square of every number from 1 to 100.

2. Write a shell script that sorts all files with names ending in ".txt" one at a time, removes duplicate entries, and saves the output to filename.txt.sorted. The script then merges all the sorted text into a single file called combined.txt.sorted. The sort can also merge presorted files when used with the -m flag.

   The standard Unix sort can be used to sort an individual file. The uniq command will remove duplicate lines that are adjacent to each other. (Hence, the data should be sorted already.)

3. Do the examples for shell loops above give you any ideas about using multiple computers to speed up processing?

8.16 Generalizing Your Code

All programs and scripts require input in order to be useful.

Inputs commonly include things like scalar parameters to use in equations and the names of files containing more extensive data such as a matrix or a database.

8.16.1 Hard-coding: Failure to Generalize

All too often, inexperienced programmers provide what should be input to a program by hard-coding values and filenames into their programs and scripts:

#!/bin/csh

# Hard-coded values 1000 and output.txt
calcp1 1000 > output.txt

Many programmers will then make another copy of the program or script with different constants or filenames in order to do a different run. The problem with this approach should be obvious. It creates a mess of many nearly identical programs or scripts, all of which have to be maintained together. If a bug is found in one of them, then all of them have to be checked and corrected for the same error.
8.16.2 Generalizing with User Input

A better approach is to take these values as input:

```
#!/bin/csh
printf "How many iterations? "
set iterations = "$<"
printf "Output file? "
set output_file = "$<"
calcpi $iterations > $output_file
```

If you don’t want to type in the values every time you run the script, you can put them in a separate input file, such as "input-1000.txt" and use redirection:

```
shell-prompt: cat input-1000.txt
1000
output-1000.txt
shell-prompt: calcpi-script < input-1000.txt
```

This way, if you have 50 different inputs to try, you have 50 input files and only one script to maintain instead of 50 scripts.

8.16.3 Generalizing with Command-line Arguments

Another approach is to design the script so that it can take command-line arguments, like most Unix commands. Using command-line arguments is quite simple in most scripting and programming languages.

In all Unix shell scripts, the first argument is denoted by the special variable $1, the second by $2, and so on.

$0 refers to the name of the command as it was invoked.

**Bourne Shell Family**

In Bourne Shell family shells, we can find out how many command-line arguments were given by examining the special shell variable " $#". This is most often used to verify that the script was invoked with the correct number of arguments.

```
#!/bin/sh

# If invoked incorrectly, tell the user the correct way
if [ $# != 2 ]; then
    printf "Usage: $0 iterations output-file\n"
    exit 1
fi

# Assign to named variables for readability
iterations=$1
output_file="$2"    # Filename may contain white space!

calcpi $iterations > "$output_file"
```

```
shell-prompt: calcpi-script
Usage: calcpi-script iterations output-file
shell-prompt: calcpi-script 1000 output-1000.txt
shell-prompt: cat output-1000.txt
3.141723494
```
C shell Family

In C shell family shells, we can find out how many command-line arguments were given by examining the special shell variable "$#argv".

```csh
#!/bin/csh

# If invoked incorrectly, tell the user the correct way
if ( $#argv != 2 ) then
   printf "Usage: $0 iterations output-file\n"
   exit 1
endif

# Assign to named variables for readability
set iterations=$1
set output_file="$2" # Filename may contain white space!

calcpi $iterations > "$output_file"
```

Usage: calcpi-script iterations output-file
shell-prompt: calcpi-script
shell-prompt: calcpi-script 1000 output-1000.txt
shell-prompt: cat output-1000.txt
3.141723494

8.16.4 Self-test

1. Modify the following shell script so that it takes the filename of the dictionary and the sample word as user input instead of hard-coding it. You may use any shell you choose.

```sh
#!/bin/sh

if fgrep 'abacus' /usr/share/dict/words; then
   printf 'abacus is a real word.\n'
else
   printf 'abacus is not a real word.\n'
fi
```

2. Repeat the above exercise, but use command-line arguments instead of user input.

8.17 Functions and Calling other Scripts

Most scripts tend to be short, but even a program of 100 lines long can benefit from being broken down and organized into modules.

The Bourne family shells support simple functions for this purpose.

C shell family shells do not support separate functions within a script, but this does not mean that they cannot be modularized. A C shell script can, of course, run other scripts and these separate scripts can serve the purpose of subprograms.

Some would argue that separate scripts are more modular than functions, since a separate script is inherently available to any script that could use it, whereas a function is confined to the script that contains it.

Another advantage of using separate scripts is that they run as a separate process, so they have their own set of shell and environment variables. Hence, they do not have side-effects on the calling script. Bourne shell functions, on the other hand, can modify "global" variables and impact the subsequent behavior of other functions or the main program.

There are some functions that are unlikely to be useful in other scripts, however, and Bourne shell functions are convenient in these cases. Also, it is generally easy to convert a Bourne shell function into a separate script, so there isn’t generally much to lose by using a function initially.
8.17.1 Bourne Shell Functions

A Bourne shell function is defined by simply writing a name followed by parenthesis, and a body between curly braces on the lines below:

```sh
name()
{
  commands
}
```

We call a function the same way we run any other command.

```sh
#!/bin/sh
line()
{
  printf ’-------------------------------------------------------------
’
}
line
```

If we pass arguments to a function, then the variables $1, $2, etc. in the function will be set to the arguments passed to the function. Otherwise, $1, $2, etc. will be the arguments to the main script.

```sh
#!/bin/sh
print_square()
{
  printf $((1 * 1))
}
c=1
while [ $c -le 10 ]; do
  printf "%d squared = %d\n" $c `print_square c`
c=$((c + 1))
done
```

The return statement can be used to return a value to the caller. The return value is received by the caller in $?, just like the exit status of any other command. This is most often used to indicate success or failure of the function.

```sh
#!/bin/sh
myfunc()
{
  if ! command1; then
    return 1
  fi
  if ! command2; then
    return 1
    return 0
  fi
  if ! myfunc; then
    exit 1
  fi
}
```

We can define local variables in a function if we do not want the function to modify a variable outside itself.

```sh
#!/bin/sh
8.17.2 C Shell Separate Scripts

Since C shell does not support internal functions, we implement subprograms as separate scripts. Each script is executed by a separate shell process, so all variables are essentially local to that script.

We can, of course, use the `source` to run another script using the parent shell process as described in Section 8.7. In this case, it will affect the shell and environment variables of the calling script. This is usually what we intend and the very reason for using the `source` command.

When using separate scripts as subprograms, it is especially helpful to place the scripts in a directory that is in your PATH. Most users use a directory such as `~/bin` or `~/scripts` for this purpose.

8.17.3 Self-test

8.18 Alias

An alternative to functions and separate scripts for very simple things is the `alias` command.

This command creates an alias, or alternate name for another command. Aliases are supported by both Bourne and C shell families, albeit with a slightly different syntax. They are most often used to create simple shortcuts for common commands.

In Bourne shell and derivatives, the new alias is followed by an `='. Any command containing white space must be enclosed in quotes, or the white space must be escaped with a `\`.

```bash
#!/bin/sh
alias dir='ls -als'
dir
```

C shell family shells use white space instead of an `=' and do not require quotes around commands containing white space.

```csh
#!/bin/csh
alias dir ls -als
dir
```

An alias can contain multiple commands, but in this case it must be enclosed in quotes, even in C shell.

```csh
#!/bin/csh
# This will not work:
# alias pause printf "Press return to continue..."; $<
#
# It is the same as:
```
8.19 Shell Flags and Variables

Unix shells have many command line flags to control their behavior.

One of the most popular shell flags is -e. The -e flag in both Bourne Shell and C shell cause the shell to exit if any command fails. This is almost always a good idea, to avoid wasting time and so that the last output of a script shows any error messages from the failed command.

Flags can be used in the shebang line if the path of the shell is fixed. When using #!/usr/bin/env, we must set the option using a separate command, because the shebang line on some systems treats everything after '#!/usr/bin/env' as a single argument to the env command.

```
#!/bin/sh -e
# The shebang line above is OK
```

```
#!/bin/csh -e
# The shebang line above is OK
```

```
#!/usr/bin/env bash -e
# The shebang line above is invalid on some systems and may cause
# an error such as "bash -e: command not found"
```

We can get around this in Bourne family shells using the set command, which can be used to turn on or off command-line flags within the script. For example, "set -e" in a script causes the shell running the script to terminate if any subsequent commands fail. A "set +e" turns off this behavior.

```
#!/usr/bin/env bash
# Enable exit-on-error
set -e
```

Unfortunately, C shell family shells do not have anything comparable to the Bourne shell set command. Recall that C shell has a set command, but it is used to set shell variables, not command-line flags.

Many features controlled by command-line flags can also be set within a C shell script using special shell variables, but -e is not one of them.

The -x flag is another flag common to both Bourne Shell and C shell. If causes the shell to echo commands to the standard output before executing them, which is often useful in debugging a script that it failing at an unknown location.

```
#!/bin/sh -x
```

```
#!/bin/csh -x
```
As stated in Section 8.6, Bourne shell family scripts do not source any startup scripts by default. Bourne shells only source files like .shrc, .bashrc, etc. if the shell is interactive, i.e. the standard input is a terminal.

C shell and T shell scripts, on the other hand, will source .cshrc or .tcshrc by default. This behavior can be disabled using the `-f` flag. Disabling this is usually a good idea, since the script may behave differently for different people, depending on what’s in their .cshrc.

There are many other command-line flags and corresponding C shell variables. For more information, run "man sh", "man csh", etc.

8.20 Arrays

Bourne shell does not support arrays, but some commands can process strings containing multiple words separated by white space.

C shell supports basic arrays. One advantage of this is that we can create lists of strings where some of the elements contain white space.

An array constant is indicated by a list enclosed in parenthesis.

Each array element is identified by an integer subscript.

We can also use a range of subscripts, separated by `-`.
The foreach command is not designed to work with arrays. It is designed to break a string into white space-separated tokens. Hence, given an array, foreach will view it as one large string and then break it wherever there is white space, which could break individual array elements into multiple pieces.

The $argv variable containing command-line arguments is an array. Hence, the $#argv variable is not special to $argv, but just another example of referencing the number of elements in an array.

8.21 Good and Bad Practices

A very common and very bad practice in shell scripting is checking the wrong information to make decisions. One of the most common ways this bad approach is used involves making assumptions based on the operating system in use.

Take the following code segment, for example:

```bash
if [ 'uname' == 'Linux' ]; then
    compiler='gcc'
    endian='little'
fi
```

Both of the assumptions made about Linux in the code above were taken from real examples!

Setting the compiler to 'gcc' because we’re running on Linux is wrong, because Linux can run other compilers such as clang or icc. Compiler selection should be based on the user’s wishes or the needs of the program being compiled, not the operating system alone.

Assuming the machine is little-endian is wrong because Linux runs on a variety of CPU types, some of which are big-endian. The user who wrote this code probably assumed that if the computer is running Linux, it must be a PC with an x86 processor, which is not a valid assumption.

There are simple ways to find out the actual endianness of a system, so why would we instead try to infer it from an unrelated fact?? We should instead use something like the open source `endian` program, which runs on any Unix compatible system.

```bash
if [ 'endian' == 'little' ]; then
fi
```

8.22 Here Documents

We often want to output multiple lines of text from a script, for instance to provide detailed instructions to the user. For instance, the output below is a real example from a script that generates random passphrases.

```bash
===========================================================================
If no one can see your computer screen right now, you may use one of the
suggested passphrases about to be displayed. Otherwise, make up one of
your own consisting of three words separated by random characters and
modified with a random capital letters or other characters inserted.
===========================================================================
```

We could output this text using six printf statements. This would be messy, though, and would require quotes around each line of text.

We could also store it in a separate file and display it with the `cat` command:

```
#!/bin/sh -e

cat instructions.txt
```
This would mean keeping track of multiple files, however.

A "here document", or "heredoc", is another form of redirection that is typically only used in scripts. It essentially redirects the standard input to a portion of the script itself. The general form is as follows:

```
command << end-of-document-marker
end-of-document-marker
```

The end-of-document-marker can be any arbitrary text that you choose. This allows the text to contain literally anything. You simply have to choose a marker that it not in the text you want to display. Common markers are EOM (end of message) or EOF (end of file).

Heredocs can be used with any Unix command that reads from standard input, but are most often used with the `cat` command:

```
#!/bin/sh -e

cat << EOM
===========================================================================
If no one can see your computer screen right now, you may use one of the
suggested passphrases about to be displayed. Otherwise, make up one of
your own consisting of three words separated by random characters and
modified with a random capital letters or other characters inserted.
===========================================================================
EOM
```

Heredocs can also be used to create files from a template that uses shell or environment variables. Any variable references that appear within the text of a heredoc will be expanded. The output of any command reading from a heredoc can, of course, be redirected to a file or other device.

```
#!/bin/csh -ef

# Generate a series of test input files with difference ending values
foreach end_value (10 100 1000 10000)
  foreach tolerance (0.0001 0.0005 0.001)
    cat << EOM > test-input-$end_value-$tolerance.txt
      start_value=1
      end_value=$end_value
      tolerance=$tolerance
    EOM
  end
end
```

### 8.23 Common Unix Tools Used in Scripts

It is often said that most Unix users don’t need to write programs. The standard Unix commands contain all the functionality that a typical user needs, so they need only learn how to use the commands and write simple scripts to utilize them.

The sections below introduce some of the popular tools with the sole intention of raising awareness. The details of these tools would fill a separate book by themselves, so we will focus on simple, common examples here.

#### 8.23.1 Grep

The `grep` command, short for General Regular exPressions, is a powerful tool for searching the content of text files.

Regular expressions are a standardized syntax for specifying patterns of text. They are similar to the globbing patterns discussed in Section 7.8.5, but the details are quite different. Also, while globbing patterns are meant to match filenames, regular expressions are meant to match strings in any context.

Some of the more common regular expression features are shown in Table 8.9.
### Token Matches

- Any character
- Any single character in list
- Any single character between first and last, in the order they appear in the character set in use. This may be affected by locale settings.
- Zero or more of the preceding token
- One or more of the preceding token

<table>
<thead>
<tr>
<th>Token</th>
<th>Matches</th>
</tr>
</thead>
<tbody>
<tr>
<td>.</td>
<td>Any character</td>
</tr>
<tr>
<td>[list]</td>
<td>Any single character in list</td>
</tr>
<tr>
<td>[first-last]</td>
<td>Any single character between first and last, in the order they appear in the character set in use. This may be affected by locale settings.</td>
</tr>
<tr>
<td>*</td>
<td>Zero or more of the preceding token</td>
</tr>
<tr>
<td>+</td>
<td>One or more of the preceding token</td>
</tr>
</tbody>
</table>

Table 8.9:

**Note** To match any special character, such as `.`, or `[`, precede it with a `\`.

On BSD systems, a POSIX regular expression reference is available via `man re_format`.

On Linux systems, a similar document is available via `man 7 regex`.

Regular expression pattern matching can be used in any language. At the shell level, patterns are typically matched using the `grep` command.

In short, `grep` searches a text file for patterns specified as arguments and prints matching lines.

```
grep pattern file-spec
```

**Note** Patterns passed to `grep` should usually be hard-quoted to prevent the shell from interpreting them as globbing patterns or other shell features.

```
# Show lines in Bourne shell scripts containing the string "printf"
grep printf *.sh

# Show lines in C programs containing strings that qualify as variable names
grep '[A-Za-z_] [A-Za-z_0-9]*' *.c

# Show lines in C programs containing decimal integers

grep '[0-9]+' *.c

# Show lines in C programs containing real numbers

grep '[0-9]*\.[0-9]+' *.c
```

By default, the `grep` command follows an older standard for traditional regular expressions, in order to maintain backward compatibility in older scripts.

To enable the newer extended regular expressions, use `grep -E` or `egrep`.

To disable the use of regular expressions and treat each pattern as a fixed string, use `grep -F` or `fgrep`. This is sometimes useful for better performance or to eliminate the need for `\` before special characters.

### 8.23.2 Stream Editors

Stream editors are a class of programs that take input from one stream, often standard input, modify it in some way, and send the output to another stream, often standard output.

The `sed` (Stream EDitor) command is among the most commonly used stream editing programs. The `sed` has a variety of capabilities for performing almost any kind of changes you can imagine. Most often, though, it is used to replace text matching a regular expression with something else. Our introduction here will focus on this feature and we will leave the rest for tutorials dedicated to `sed`.

The basic syntax of `sed` command for replacing text is as follows:
The `sed` command is a powerful stream editor that is used to modify text streams. It is similar to `grep` but offers more complex text manipulation capabilities.

```
sed -e 's|pattern|replacement|g'
```

The `−e` flag specifies the use of traditional regular expressions. To use the more modern extended regular expressions, use `−E` as with `grep`.

The `'s'` is the `substitute` command. Other commands, not discussed here, include 'd' (delete) and 'i' (insert).

The '|' is a separator. You can use any character as the separator as long as all three separators are the same. This allows any character to appear in the pattern or replacement text. Just use a separator that is not in either. The most popular separators are '|' and '/', since they usually stand out next to typical patterns.

The pattern is a regular expression, just as we would use with `grep`. Again, special characters that we want to match literally must be escaped (preceded by a '\').

The replacement text is not a regular expression, but may contain some special characters specific to `sed`. The most common is '&', which represents the current string matching the pattern. This feature makes it easy to add text to strings matching a pattern, even if they are not the same.

The `−g` means perform a global replacement. If omitted, only the first match on each line is replaced.

```
# Get snooty
sed -e 's|Bob|Robert|g' file.txt > modified-file.txt

# Convert integer constants to long constants in a C program
sed -e 's|[0-9]+|[&L]|g' prog.c > prog-long.c
```

The `tr` (translate) command is a simpler stream editing tool. It is typically used to replace or delete individual characters from a stream.

```
# Capitalize all occurrences of 'a', 'b', and 'c'
tr 'abc' 'ABC' file.txt > file-caps.txt

# Delete all digits from a file
tr -d '0123456789' file.txt > file-qless.txt
```

### 8.23.3 Tabular Data Tools

Unix systems provide standard tools for working with tabular data (text data organized in columns).

The `cut` command is a simple tool for removing a portion from each line of a text stream. The user can specify byte, character, or field positions to be removed.

```
# Remove the 3rd and 4th characters from every line
cut -c 3-4 file.txt > chopped-file.txt

# Remove the first column of numbers separated by white space
cut -w -f 1 results.txt > results-without-col1.txt
```

The `awk` command is an extremely sophisticated tool for manipulating tabular data. It is essentially a non-interactive spreadsheet, capable of doing modifications and computations of just about any kind.

Awk includes a scripting language that looks very much like C, with many extensions for easily processing textual data.

Entire books are available on `awk`, so we will focus on just a few basic examples.

`Awk` is generally invoked in one of two ways. For very simple awk operations (typically 1-line scripts), we can provide the awk script itself as a command-line argument, usually hard-quoted:

```
awk [-F field-separator] 'script' file-spec
```

For more complex, multi-line scripts, it may prove easier to place the awk script in a separate file and refer to it in the command:
Input is separated into fields by white space by default, but we can specify any field-separator we like using the `-F`. The field separator can also be changed within the awk script by assigning the special variable FS.

Statements within the awk script consist of a pattern and an action.

Patterns may be relational expressions comparing a given field (column) to a pattern. In this case, the action will be invoked only on lines matching the pattern.

If pattern is omitted, the action will be performed on every line of input.

The special patterns BEGIN and END are used to perform actions before the first line is processed and after the last line is processed.

The action is essentially a C-like function. If omitted, the default action is to print the entire line matching pattern. (Hence, awk can behave much like grep.)

Example 1: A simple awk command

```
# Print password entries for users with uid >= 1000
shell-prompt: awk -F ':' '$3 >= 1000 { print $0 }' /etc/passwd
nobody:*:65534:65534:Unprivileged user:/nonexistent:/usr/sbin/nologin
joe:*:4000:4000:Joe User:/home/joe:/bin/tcsh
```

Example 2: A separate awk script

```
# Initialize variables
BEGIN {
    sum1 = sum2 = 0.0;
}

# Add column data to sum for each line
{
    print $1, $2
    sum1 += $1;
    sum2 += $2;
}

# Output sums after all lines are processed
END {
    printf("Sum of column 1 = %f\n", sum1);
    printf("Sum of column 2 = %f\n", sum2);
}
```

```
shell-prompt: cat twocol.txt
4.3 -2.1
5.5 9.0
-7.3 4.6

shell-prompt: awk -f ./sum.awk twocol.txt
4.3 -2.1
5.5 9.0
-7.3 4.6
Sum of column 1 = 2.500000
Sum of column 2 = 11.500000
```

8.23.4 Sort/Uniq

The sort command is a highly efficient, general-purpose stream sorting tool. It sorts the input stream line-by-line, optionally prioritizing the sort by one or more columns.
shell-prompt: cat names.txt
Kelso Bob
Cox Perry
Dorian John
Turk Christopher
Ried Elliot
Espinosa Carla

# Sort by entire line
shell-prompt: sort names.txt
Cox Perry
Dorian John
Espinosa Carla
Kelso Bob
Ried Elliot
Turk Christopher

# Sort by second column
shell-prompt: sort -k 2 names.txt
Kelso Bob
Espinosa Carla
Turk Christopher
Ried Elliot
Dorian John
Cox Perry

Shell-prompt: cat numbers.txt
45
-12
32
16
7
-12

# Sort sorts lexically by default
Shell-prompt: sort numbers.txt
-12
-12
16
32
45
7

# Sort numerically
Shell-prompt: sort -n numbers.txt
-12
-12
7
16
32
45

The **uniq** command eliminates adjacent duplicate lines from the input stream.

Shell-prompt: uniq numbers.txt
45
-12
32
16
7
-12
8.23.5 Perl, Python, and other Scripting Languages

All of the commands described above are described by the POSIX standard and included with every Unix compatible operating system.

A wide variety of tasks can be accomplished without writing anything more than a shell script utilizing commands like these. Nevertheless, some Unix users have felt that there is a niche for tools more powerful than shells scripts and standard Unix commands, but more convenient than general-purpose languages like C, Java, etc.

As a result, a new class of scripting languages has evolved that are somewhat more like general-purpose languages. Among the most popular are TCL, Perl, PHP, Python, Ruby, and Lua.

These are interpreted languages, so performance is much slower than a compiled language such as C. However, they are self-contained, using built-in features or library functions instead of relying on external commands such as awk and sed. As a result, many would argue the they are more suitable for writing sophisticated scripts that would lie somewhere between shell scripts and general programs.
Chapter 9

Data Management

9.1 What is Data Management?

Data management, in the context of research computing, refers to how research data is stored, formatted, and disseminated over the long term.

Researchers who generate new data need to plan ahead in order to ensure that any data supporting their research conclusions will be available in the future. The data could be used to reproduce or otherwise verify the research results, or could be analyzed in new ways for completely different purposes.

9.2 UWM Data Management Services

The UWM Libraries offer assistance with data management and development of data management plans required by many funding organizations.

For more information, visit http://www4.uwm.edu/libraries/dataservices/.

9.3 Why Worry?

Confidence in science is based on transparency, and the ability to reproduce results by repeating experiments. Without access to the raw data generated by, or used in your experiments, your experiments cannot be verified by others.

Given the often high cost of generating data, preserving data for future use can often prove to be far more cost-effective.

The National Science Foundation is now requiring all applicants to submit a detailed plan for preserving and disseminating their research data. I.e., you will not be rewarded a grant from the NSF unless you clearly state your plans for long-term data management.

9.4 Storage Logistics

There are many issues you will need to consider in order to plan well for data management. A few of those issues are discussed in the sections that follow.

9.4.1 Data Format

When storing data only for yourself, you might not give this issue much thought. However, data management includes not only preservation, but dissemination. If others will have access do your data, it must be in a format that is easy for them to read.
Many areas of science have developed standard data formats to help researchers and software applications interoperate. Some common data formats are netCDF (Network Common Data Format) is a generic format for any array-oriented data. NIfTI is an imaging data format used in MRI and other imaging research.

Research needs are too diverse to cover all of the standard data formats here. The goal of this section is to raise awareness and encourage researchers to explore available standards before digging themselves into a hole.

---

**Caution** Changing the format of large amounts of your data at a later time could be a very frustrating and costly process. It is highly advisable to decide on a standard data format before your research progresses too far.

---

The best way to explore data formats is by talking to others in your specific field. This will help you develop a sense of what the emerging standards are in your niche.

### 9.4.2 Lifespan

Another very important question to ask is how long the data should be preserved. This will impact the cost of data management, although not as much as you might think, assuming that storage costs continue to decline over the long term.

Generally, the harder it is to regenerate the data, the longer it should be preserved.

### 9.4.3 Security

If the data contain confidential information, such as personal health information (PHI) or financial records, it may be necessary to restrict and track access to it. Regulations on PHI data are strict and somewhat complex, so they should be explored before making any data management plans.

### 9.4.4 Safety

Data safety refers to the risk of data loss. If you’re using a service provider to store your data, this will generally be their responsibility. They will maintain backups of data they store and provide a written guarantee about its availability.

If you are storing the data yourself, you’ll need to think about how to back it up and where. Backups should always be stored far from the original data in order to protect against fire, theft, and other physical disasters.

### 9.4.5 Funding

Paying for long-term data storage is a complex issue. Depending on the cost involved, it may or may not be possible to pay for it from a one-time grant allocation. Some institutions may provide data storage services, but in most cases, researchers will have to make their own arrangements.

### 9.4.6 Storage Providers

There are a number of organizations that provide long-term data storage, provided by Universities, government organizations, and private companies.

Some examples include the NIH’s GenBank, DataSpace, Amazon Simple Storage Service (S3), Google Cloud Storage, and DuraSpace.

Again, the best approach to selecting one is investigating their *current* service offerings and talking to colleagues who have been down this road.
9.5 Data Storage on the Cluster

Storage on clusters is generally designed for speed, not long-term capacity or data safety. Like most clusters, our cluster has a fairly large RAID storage pool which is reasonably fast, but is not backed up.

Data storage on UWM’s clusters should be considered temporary space. Space is limited, and users are constantly generating a lot of new data. Hence, it is important for all users to move data off the cluster as soon as possible in order to keep the cluster storage available for other jobs.

This does not mean that you cannot leave data on the cluster for further analysis. However, all data generated on the cluster should be copied to another location immediately after being generated, so that it will be safe from disasters such as hardware failures or fires. It should be removed from the cluster as soon as it is safely stored in two other locations where it is accessible for further analysis.

9.6 Data Transfer

Storage is not the only problem associated with big data. It also presents challenges with transferring data, especially over great distances and across different computer platforms.

This can be particularly problematic for small organizations that do not have a very high bandwidth Internet connection. While the Internet backbone may provide plenty of speed to transfer your research data in a reasonable amount of time, the connection from the Internet into your building may be a severe bottleneck. This is known as the "last mile" problem.

One potential solution to this problem is to avoid transferring the data in the first place. Some organizations offer web-based tools to allow users elsewhere to perform common analyses on their data without first downloading it. Rather than moving the data to the software, the software, which is much smaller, is moved to the data.

Another potential solution is to simply perform a more selective transfer. Determining exactly which parts of the data to transfer can involve a lot of manual labor, however.

Sometimes the problem is not bandwidth, but user interface. The most common type of data transfer utilized ordinary tools like a web browser or FTP client. These methods are collectively known as "data schlepping".

Data schlepping requires the user to use a variety of tools to transfer data to and from various sites. It also often suffers from failures due to dropped network connections, power outages, and other issues that are likely to interrupt a long transfer. Some tools, such as rsync, allow an interrupted transfer to continue from where it left off. However, not all sites offer rsync service.

Globus Transfer is an example of a web-based alternative for data transfer that has built-in capabilities for dealing with connection issues, login credentials, and many other data transfer issues.

Data transfer tools are evolving rapidly in response to the growing needs presented by big data. Users should make it a habit to continuously explore and reevaluate new and existing options.
Part II

Parallel Computing
Chapter 10

Parallel Computing Overview

10.1 Motivation

In 1976, Los Alamos National Laboratories purchased a Cray-1 supercomputer for $8.8 million. As the world’s fastest computer at the time, it had 8 mebibytes (a little over 8 million bytes) of main memory and was capable of 160 million floating point operations per second. (Source: http://www.cray.com/About/History.aspx)

The first release of this manual was written in September 2010 on a $700 desktop computer with a gibibyte (a little over 1 billion bytes) of main memory, and capable of over 2 billion floating point operations per second.

It may seem that today’s computers have made the supercomputer obsolete. However, there are still, and always will be many problems that require far more computing power than any single processor can provide.

There are many examples of highly optimized programs that take months to run even with thousands of today’s processors. The volume and resolution of raw data awaiting analysis has exploded in recent years due to advances in both research techniques and technology. Enormous amounts of new data are being generated every day, and new ways to analyze old data are constantly being discovered.

It is important to understand the difference between parallel computing and parallel programming. Parallel computing includes any situation where multiple computations are being done at the same time.

This often involves running multiple instances of the same serial program at the same time, each with different inputs. Typically, there is no communication or cooperation between the multiple instances as they run. This scenario is known as embarrassingly parallel computing.

Parallel programming, on the other hand, involves writing a special program that will utilize multiple processors. The code running on each processor will exchange information with the others as they all run. This is much more complex than embarrassingly parallel computing, but is necessary for many problems.

There are many types of parallel architectures, and each is suited for specific types of problems. Writing parallel programs is not a process that can be easily generalized. Understanding of specific algorithms is crucial in determining if and how they can be decomposed into independent subtasks, and what will be the most suitable parallel architecture on which to run them. Some of the common parallel architectures are outlined in the following sections.

10.2 Shared Memory and Multi-Core Technology

Processor manufacturers have recently found it difficult to continue increasing processor clock speeds at the rate we’ve become accustomed to in the past decade or two. While this hurdle will certainly be overcome eventually, the industry has realized that processor clock speeds cannot grow indefinitely, and have therefore turned their focus toward improving efficiency and parallelism.

As a result, most personal computers now come with multiple cores. Core is the modern term for what has been traditionally known as the Central Processing Unit (CPU) or simply processor. The term core refers to a functional CPU. For a long time
before the age of multi-core technology, processor chips traditionally contained a single CPU. Hence, the term CPU has been widely used to refer to either the functional CPU or the physical chip. Now that the assumption of one CPU per chip is no longer valid, the term CPU has become somewhat ambiguous, and the term core is preferred when referring to a functional CPU.

Each core is capable of running its own thread, or sequence of instructions. Hence, a multi-core CPU chip can run multiple programs at the same time, or multiple instructions of the same program, provided that one such instruction doesn’t need to use the results of another.

Since these multiple cores are part of the same computer, they have access to the same memory. This allows the multiple threads to share information very easily. The disadvantage of shared memory is that only one core can access a memory chip at a given time. Therefore, the number of cores that can truly work in parallel is very limited, i.e. shared-memory architectures don’t scale well. There are techniques in use to reduce memory contention, such as dividing memory into multiple independent banks, caching, and so on. Generally speaking, however, shared memory parallelism is only effective on a small number of cores on commodity hardware. Larger scale shared memory is only feasible on specialized (and expensive) supercomputers.

10.3 Distributed Parallel Computing

10.3.1 Clusters and HPC

In order to achieve higher degrees of parallelism, work must be distributed among independent processors that don’t contend for shared resources. Instead of sharing memory, processes in a distributed system run on separate computers that each have their own local memory. The processes communicate with each other by passing messages over a high-speed network. Message passing is not as easy to program or as fast as shared memory, but it does allow for much greater numbers of cores to be used by cooperating processes in many cases. In other words, it scales better than shared-memory parallelism.

If it’s possible to decompose a large computational job into somewhat independent sub-tasks, a cluster can provide a huge speed-up by deploying the work to hundreds or even thousands of cores at the same time. This type of computing is often referred to as high performance computing (HPC).

A cluster is a collection of essentially ordinary computers with a dedicated high-speed network and special software to facilitate job scheduling and communication. Each computer in the cluster, called a node, is typically an off-the-shelf personal computer with a fast processor and lots of RAM, owing to the heavily computational nature of the programs typically run on clusters.

Most clusters also offer shared disk space, so that the same files are directly accessible to all nodes in the cluster. The shared space is a more or less typical file server shared by all the nodes, although on large clusters, it may be implemented using a different special file system designed specifically for HPC.

As stated earlier, clusters are suitable for problems that can be decomposed into a large number of relatively independent tasks that can be distributed among multiple computers to run in parallel. If decomposing a problem results in tasks that must share extremely large volumes of data, then a shared memory architecture may provide better performance. Fortunately, there are many large computational problems that adapt well to the distributed computing model offered by clusters.

There are several types of nodes in a typical cluster:

- The head node is responsible for running the job scheduler and possibly other system tasks.
- A login node is where users log in to run Unix shell commands. Users typically use the login node to edit scripts, submit jobs to the scheduler, and monitor their jobs. On a small to medium sized cluster, the head node typically serves as the login node. Very large clusters may provide one or more login nodes separate from the head node.
- Compute nodes run the processes that make up scheduled jobs. Most of the nodes in a cluster are compute nodes. Compute nodes typically have faster processors and more RAM than head and login nodes.
- I/O nodes are the file servers in a cluster. I/O nodes typically run NFS (the Unix Network File system Service) on small and medium clusters, or a more sophisticated parallel file system on large clusters. In either case, the login nodes and compute nodes all have access to the files on the I/O nodes.
- A visualization node is another node that users can log into in order to run shell commands. However, a visualization node is meant to run graphical software for viewing the results of jobs run on a cluster.
Note that using a visualization node usually involves rendering graphics over a network, i.e. the graphical program is running on the visualization node and displaying on a different machine, probably your workstation, through the network. This will be slower than running a graphical program on your workstation and displaying directly to the attached monitor. However, it may save time if it eliminates the need to frequently download large files to be visualized.

10.3.2 Grids and HTC

A grid is similar to a cluster, but without the high-speed communication network and high-speed shared storage typically seen on clusters. Grids often utilize hardware that was not designed and built primarily for parallel computing. Instead, grids are commonly implemented to take advantage of already existing hardware that may not otherwise be well utilized, such as PCs in college computer labs. Lab PCs tend to be heavily utilized occasionally (during classes) but idle most other times.

Grids are often implemented with no improvements to the existing hardware infrastructure. Since the computers are already installed and connected by a network, a grid can be often implemented simply by installing and configuring grid management software such as the University of Wisconsin’s HTCondor system, available at http://research.cs.wisc.edu/htcondor/. It is not even necessary for all the computers to run the same operating system, although standardizing will make the grid easier to use. The standardization of PC configurations can be rendered unnecessary using a system like Marquette University’s Pools Of Virtual Boxes (POVB), which implements a grid of identical virtual Linux machines running on Windows lab PCs that may be running different versions of Windows. For more information, visit the POVB website at http://www.marquette.edu/mugrid/povb.shtml.

Since machines on a grid communicate over a standard, non-dedicated campus or office network, extensive message-passing between parallel processes would be likely to overload the network. Therefore, some of the programs that we typically run on clusters are not suitable for grids.

Grids are only suitable for algorithms that can be decomposed into highly independent processes that require little or no communication with each other while running, i.e. embarrassingly parallel computing. Grid users often simply run many instances of serial programs at the same time. In other words, grid users often do parallel computing without parallel programming.

Grid computing is often referred to as High Throughput Computing, or HTC, since the lack of communication between processes eliminates potential bottlenecks that might slow the progress of the computations.

Many campuses are implementing grids to provide an inexpensive parallel computation resource for students and researchers. One of the leaders in this area is Purdue University, where every computer on the entire campus is part of a massive grid.

Note that HTC can and often is done on clusters as well as grids.

10.4 Other Parallel Architectures

10.4.1 Graphics Processors and CUDA

Today’s video processors, also known as graphics processing units (GPUs), require an enormous amount of computing power in order to quickly render the rapidly changing 3-dimensional graphics of animation software. However, like the CPUs in lab PCs, GPUs tend to sit idle much of the time on a typical PC. Software has been recently developed to utilize the significant processing power of GPUs when they are not being used for graphics rendering. CUDA is a free, open source system for utilizing several models of video processors. Using a system such as CUDA, programmers can boost the performance of programs by running computations in parallel on the CPUs and the GPUs.

One caveat is that while GPUs have a great deal of processing power, they are designed specifically for graphics rendering, and are therefore not well suited to every computational application. The interface to a GPU may lend itself very well to some applications, and not so well to others. The subject is complex, and readers are advised to research whether GPUs would fit their needs before investing in new hardware.

Nevertheless, some sites have invested in clusters with large numbers of GPUs in order to gain the best performance for certain applications that lend themselves well to GPU computing. A fair number of GPU-based programs have already been developed and having access to a pool of GPUs allows such software to be easily utilized.
10.4.2 Vector Processors

Before the appearance of fast and inexpensive microprocessors, supercomputers were often designed that allowed a single instruction to perform the same operation on many data elements at the same time. This type of single-instruction, multiple data (SIMD) parallelism is useful for many vector and matrix manipulations, and is very easy to program compared to most other types of parallelism. SIMD machines are less popular now, due to the availability of cheaper solutions such as clusters and grids. However, they still have a role to play for solving important problems that are not suitable for other parallel architectures.

10.5 Best Practices in Parallel Computing

10.5.1 Parallelize as a Last Resort

While there will always be a need for parallel computing, the availability of parallel computing resources may tempt people to use them as a substitute for writing good code. There is virtually no optimal software in existence. Most software in existence at any given moment can be made to run faster, and a significant percentage of it can be made to run orders of magnitude faster. Most performance issues can and should therefore be resolved by optimizing the software first.

Improving software is a more intelligent way to resolve performance issues wherever it’s possible. It will allow effective use of the software on a much wider variety of hardware, possibly including ordinary desktop and laptop machines. This is a much better situation than needing a cluster or grid to get your work done.

It is also the more ethical way to resolve issues where users are running on shared resources. Using tens of thousands of dollars worth of computer equipment to make inefficient software run in a reasonable amount of time is wasteful and foolish, and may delay the work of others who need those resources for more intelligent uses.

Before you consider the use of parallel computing, make sure you’ve done all you can to optimize your software, by choosing efficient algorithms, using compiled languages for the time-consuming parts of your code, and eliminating wasteful code. Software performance is discussed in greater detail in Part III.

10.5.2 Make it Quick

Parallel computing jobs should be designed to finish as quickly as possible, preferably within a few hours or a few days. Running processes much less than a few hours is generally not practical due to the high overhead and limited benefit.

Jobs that run for weeks or months have a lower probability of completing successfully. The longer individual processes run, the higher the risk risk of being interrupted by power failures, hardware issues, security updates, etc. In the case of HTCondor grids, jobs that run for more than a few hours run a high risk of being evicted from desktop machines that are taken over by a local user.

Shorter running jobs also give the scheduler the ability to balance the load fairly among many users.

In the case of HTC (embarrassingly parallel computing), we can usually break up the work into as many pieces as we like. For example, we may have 3,000 hours with of computation that we can run as a hundred 30-hour processes, or a thousand 3-hour processes, simply by dividing up the input into smaller chunks.

There are some parallel jobs that don’t scale well, such as shared-memory or certain MPI parallel programs. In these cases, we may not be able to speed things up by simply using more cores. We may be able to divide the work into multiple stages, though, rather than one huge multistage analysis.

We should make every effort to avoid long-running jobs, though, for own sake and the sake of other users.

10.5.3 Don’t Hog the Cluster (or Grid)

If you need to submit a large batch of computation, keep a close eye on cluster load and be prepared to cancel some of it if other users need the resources.

If your cluster or grid has a priority mechanism, submit some of your jobs with lower priority, so that they can be automatically preempted when the load gets high.
10.6 Development and Testing Servers

A well-organized software development operation consists of up to five separate environments (tiers) for developing, testing, and ultimately using new software.

1. Development
2. Testing
3. Quality assurance
4. Staging
5. Production

A detailed description of the five tiers can be found in the Wikipedia article on Deployment Environments. Typically only organizations with a large software development staff will employ all five tiers.

Even in the smallest environments, though, a clear distinction is made between development/testing servers and production servers.

Virtually all clusters and grids are meant to be production environments. A cluster or grid is not a good place to develop or test new programs, for multiple reasons:

• All jobs run on a cluster or grid must go through a scheduler, which makes the development and testing process more cumbersome.

• We generally don’t want test runs of unfinished or unfamiliar code competing with production jobs. Bugs in the code or mistakes in using it will often have unforeseen impact on the system, which may harm important production jobs.

• A cluster or grid is not necessary for testing code, even parallel programs. Most testing of any program, including MPI programs, can be done on a single computer, even with a single core.

These principles apply whether you are developing your own code or learning to use a program written by others.

Code should be developed and tested on servers completely separate from the scheduled production environment of a cluster or grid.

This is easily done using a separate server with the same operating system and software installed as a compute node on the production cluster or grid.

Think of development server as a compute node where you are free from the need to schedule jobs and worries about impacting the important work of other users.

Here you can quickly and easily make changes to your program and run small test cases. Once you are confident that the code is working properly, you can move up to the next available tier for additional testing or production runs.
Chapter 11

Job Scheduling

Before You Begin
Before reading this chapter, you should be familiar with basic Unix concepts (Chapter 7), the Unix shell (Section 7.4.3, redirection (Section 7.17.1), and shell scripting (Chapter 8).

11.1 Fair Share Guidelines

Caution

BEFORE you start submitting jobs to a cluster or grid, you MUST know how to monitor and control them. Jobs that go rogue can cause problems for other users, so you need to know how to watch over them to ensure they’re not using more resources than expected.
If a job does go astray, you need to know how to terminate it quickly, before it impacts other users.

If a job is submitted and the resources required to run it are not available, the job waits in a queue until the resources become available. These pending jobs are, for the most part, started in the order they were submitted. Hence, if one user submits too many jobs at once, other users’ jobs may end up waiting in the queue for a very long time.

As a general rule, if you are already using a significant share of the total cluster resources and there are jobs pending, you should not submit new jobs until other users’ jobs have a chance to run.

If there are a lot of idle resources on the cluster, it’s generally better to utilize them by submitting some "extra" jobs than to let them remain idle. The ideal state for a cluster is nearly 100% utilized, but without jobs pending for a long time.

However, keep in mind that other users may need resources after you’ve submitted. Hence, if you’re using a lot of resources, please check the cluster load once per hour. If pending jobs appear after you’ve submitted, and none of your jobs are near completion, please kill some of them to allow other users a fair share of cluster resources.

11.2 Remote Execution

Since the early days of Unix, it has been possible to run commands on remote computers over a network connection using remote shell programs such as rsh and ssh:

```
mpc: ssh mylogin@peregrine.hpc.uwm.edu ls
Password: (enter password here)
Data
Desktop
Work
```
Caution Older protocols such as rlogin, rsh, and telnet, should no longer be used due to their lack of security. These protocols transport passwords over the Internet in unencrypted form, so people who manage the gateway computers they pass through can easily read them.

On a typical network, the above command would prompt the user for a password to log into peregrine as the user bootcamp, run the `ls` command on peregrine, and then exit back to the local host from which `ssh` was run.

It is possible to configure remote computers to accept password-less logins from trusted hosts. With a password-less login, we can run a command on a remote system almost as easily as on the local system:

```bash
mypc: ssh mylogin@peregrine.hpc.uwm.edu
Password: (enter password here)
peregrine: ssh compute-01 ls
```

In the example above, the node `compute-01` does not ask for a password, since the request is coming from `peregrine`, which is a trusted host.

Key Point Password-less login makes it convenient to automate the execution of commands on many remote machines. This idea forms the basis for distributed parallel computing.

In distributed parallel computing, a job consists of multiple processes running on different computers in order to solve a single problem.

Creating a distributed parallel job could be as simple as writing a script containing several remote execution commands. However, multiple users running such jobs without some sort of traffic control could quickly lead to chaos. The solution to this problem, scheduling software, is discussed in Section 11.3.

Distributed parallel computing environments are generally divided into the following two categories:

- A grid is made up of loosely coupled computers which may be in multiple distant locations. Grids often utilize spare CPU time on existing office or lab PCs. No additional hardware is generally required to form a grid from existing PCs on a network.
A cluster is like a grid, but generally has a dedicated high-speed network and high-speed disk storage that is directly accessible to all the computers in the cluster. The dedicated network and shared disk allow processes on a cluster to communicate heavily with each other without overloading the office or campus network. The computers within a cluster are usually located in the same room, and often mounted together in a rack.

In both grids and clusters, the various computers are referred to as nodes. A user logs into one of the nodes, known as a head node or submit node, and starts a job that dispatches processes on some of the other nodes, which are known as compute nodes, using ssh or some other remote shell program.

11.2.1 Self-test

1. Show a command for doing a long-listing of the /etc directory on the host some.unix.machine.edu, for which you have an account with login name “bob”.

2. What is passwordless login?

3. Explain the difference between a cluster and a grid.

11.3 Scheduler Basics

11.3.1 Purpose of a Scheduler

Unix systems can divide CPU cores and memory resources among multiple processes. Memory is divided up physically, and CPU time is divided temporally, with time slices of about 1/100 to 1/1000 of a second given to each process in a mostly round-robin fashion. This type of CPU sharing is known as context switching or preemptive multitasking. If you run ps -ax, ps -ef or top on any Unix system, you will see that there are far more processes running than there are cores in the system.

Sharing time on individual cores works well for programs that are idle much of the time, such as word processors and web browsers that spend most of the time waiting for user input. It can create the illusion that multiple processes are actually running at the same time, since the time is takes to respond to user input may be a small fraction of a second for any one of the processes sharing the CPU.

However, this sort of multitasking does not work well for intensive computational processes that use the CPU constantly for hours, days, or weeks.

There is overhead involved in switching a core between one process and another, so the total time to finish all processes would actually be lower if we ran one process to completion before starting the next. The more frequently a core switches between
processes, the more overhead it incurs, and the longer it will take to complete all processes. More frequent context switching sacrifices overall throughput for better response times. Just how much overhead is incurred depends on how much the processes contend with each other for memory and disk resources. The difference could be marginal, or it could be huge.

There is also the possibility that there simply won’t be enough memory available for multiple processes to run at the same time. Hence, most systems that are used to run long, CPU-intensive processes employ a scheduler to limit the number of processes using the system at once. Typically, the scheduler will allow at most one process per core to be running at any given moment. Most schedulers also have features to track and balance memory use.

When you run a job under a scheduler, the scheduler locates nodes with sufficient cores, memory and any other resources your job requires. If then marks those resources as "occupied" and dispatches your processes to the nodes containing those resources. When your job completes, the resources are again marked as "available" so that they can be allocated to the next job.

### Caution
Out of courtesy to other users on a shared cluster or grid, you should never run anything that requires significant CPU time, memory, or disk I/O on the head node or any other node that users log into directly. Doing so can overload the computer can cause it to become sluggish or unresponsive for other users. It is generally OK to perform trivial tasks there such as editing programs and scripts, processing small files, etc. Everything else should be scheduled to run on compute nodes.

After submitting a job to the scheduler, you can usually log out of the head/submit node without affecting the job. Hence, you can submit a long-running job, leave, and check the results at a later time.

### Caution
When running jobs in a scheduled environment, all jobs must be submitted to the scheduler. The scheduler does not know what resources are being used by processes it did not start, so it will not be able to guarantee performance and stability of the system if anything is run outside the scheduler.

Another advantage of schedulers is their ability to queue jobs. If you submit a job at a time when insufficient resources are available, you can walk away knowing that it will begin executing as soon as the necessary resources become available.

### 11.3.2 Common Schedulers

**HTCondor**

HTCondor is a sophisticated scheduler designed primarily to utilize idle time on grids of existing personal computers across an organization. Many college campuses use HTCondor grids to provide inexpensive parallel computing resources to their users.

**Load Sharing Facility (LSF)**

Load Sharing Facility (LSF) is a proprietary scheduler primarily used on large Linux clusters. UWM’s research cluster, Avi, runs LSF.

**Portable Batch System (PBS)**

Portable Batch System is an open source scheduler used on clusters of all sizes. It is the de facto standard for cluster scheduling, and is used on the majority of XSEDE (formerly TeraGrid) sites. The PBS scheduler is currently used on the UWM student cluster, peregrine.

The most popular and modern implementation of the PBS scheduler is Torque. There is also an open source extension for Torque called Maui, which provides enhanced scheduling features, as well as a more sophisticated, commercial version called MOAB.
Simple Linux Utility for Resource Management (SLURM)

SLURM is a relatively new open source resource manager and scheduler. It was originally developed on Linux, but naturally runs on other Unix-compatible systems as well. SLURM is distinguished from other schedulers by its high throughput, scalability, modularity, and fault-tolerance.

Sun Grid Engine (SGE)

Sun Grid Engine is an open source scheduler originally developed as a proprietary product at Sun Microsystems for their Solaris Unix system. Since it was open sourced, it has become fairly popular on other Unix variants such as BSD and Linux. It is used on clusters of all sizes.

11.3.3 Job Types

Batch Serial

A batch serial job allocates one core and executes a single process on it. All output that would appear on the terminal screen is redirected to a file. You do not need to remain logged into the submit node while a batch job is running, since it does not need access to your terminal.

This is not parallel computing, but clusters and grids may be used this way just to utilize software that users may not have on their own computers. Batch serial jobs are also often used for pre-processing (prep work done before a parallel job) and post-processing (wrap-up work done after a parallel job) that cannot be parallelized.

Interactive

An interactive job is like a batch serial job, in that a single core is usually used. However, terminal output is not redirected to a file, and the process can receive input from the keyboard. You cannot log out of the submit node while an interactive job is running. Interactive jobs are rarely used, but can be useful for short-running tasks such as program compilation.

Batch Parallel (Job Arrays)

A batch parallel job runs the same program simultaneously on multiple cores, each using different inputs. The processes are generally independent of each other while running, i.e. they do not communicate with each other. Batch parallel jobs are often referred to as embarrassingly parallel, since they are so easy to implement.

Multicore

A multicore job covers all other types of jobs. Typically, a multicore job is dispatched to a single core like a batch serial job. The scheduler is asked to allocate more than one core, but not to dispatch processes to them. The scheduler only dispatches a master process to one core, and it is up to the master process to dispatch and manage processes on the other scheduled cores. Processes within a multicore job usually communicate and cooperate with each other during execution. Hence, this is the most complex type of distributed parallel programming.

Most multicore jobs use the Message Passing Interface (MPI) system, which facilitates the creation and communication of cooperative distributed parallel jobs. MPI is discussed in detail in Chapter 36.

11.3.4 Checkpointing

The longer a program runs, the more likely it is to experience a problem such as a power outage or hardware failure before it completes.

Checkpointing is the process of periodically saving the partial results and/or current state of a process as it executes. If the exact state of a process can be saved, then it can, in theory, be easily resumed from where it left off after being interrupted. If only
partial results can be saved, it may be harder to resume from where you left off, but at least there is a chance that you won’t have to start over from the beginning.

If starting over will be a major inconvenience, then checkpointing is always a good idea. How to go about checkpointing depends heavily on the code you’re running. Simple batch serial and batch parallel jobs can be checkpointed more easily than shared memory or MPI jobs. Tools exist that allow you to checkpoint simple jobs without any additional coding. More complex jobs may require adding to your code so that it checkpoints itself.

Consult the current scheduler documentation or talk to your facilitator for more information.

11.3.5 Self-test

1. What kind of problems would occur if users of a cluster or grid used remote execution directly to run jobs?

2. Why is time-sharing of cores not a good idea in a cluster or grid environment? What types of processes work well with time-sharing?

3. What does a job scheduler do? How does it solve the problem of multiple jobs competing for resources?

4. Describe several popular job schedulers and where they are typically used.

5. Describe the four common categories of jobs that run on clusters and grids.

6. What is checkpointing and what are some of its limitations?
Chapter 12

Job Scheduling with SLURM

Before You Begin
Before reading this chapter, you should be familiar with basic Unix concepts (Chapter 7), the Unix shell (Section 7.4.3, redirection (Section 7.17.1), shell scripting (Chapter 8) and job scheduling (Chapter 11).

For complete information, see the official SLURM documentation at http://slurm.schedmd.com/.

12.1 The SLURM Scheduler

12.1.1 Cluster Status

The goal of this section is to provide a quick introduction to the SLURM scheduler.

Before scheduling any jobs through SLURM, it is often useful to check the status of the nodes. Knowing how many cores are available may influence your decision on how many cores to request for your next job.

For example, if only 50 cores are available at the moment, and your job requires 200 cores, the job will have to wait in the queue until 200 cores are free. You may end up getting your results sooner if you reduce the number of cores to 50 or less so that the job can begin right away.

Partitions

The `sinfo` command shows information about the nodes in the cluster. This can be used to determine the total number of cores in the cluster, cores in use, etc.

```
shell-prompt: sinfo
PARTITION AVAIL TIMELIMIT NODES STATE NODELIST
default-partition* up infinite 2 idle compute-[001-002]

shell-prompt: sinfo --long
Wed Nov 27 13:34:20 2013
PARTITION AVAIL TIMELIMIT JOB_SIZE ROOT SHARE GROUPS NODES STATE ←
    NODELIST
default-partition* up infinite 1-infinite no NO all 2 idle ←
    compute-[001-002]

shell-prompt: sinfo -N
NODELIST NODES PARTITION STATE
compute-[001-002] 2 default-partition* idle
```
The Ganglia Resource Monitor

In addition to the command line tools used to control and monitor jobs, there are also web-based tools for monitoring clusters and grids in a more visual manner.

The Ganglia Resource Monitor is a web-based monitoring tool that provides statistics about a collection of computers on a network.

The status of the student cluster, Peregrine, can be seen at http://www.peregrine.hpc.uwm.edu/ganglia/.

The status of the research cluster, Avi, can be seen at http://www.avi.hpc.uwm.edu/ganglia/.

12.1.2 Job Status

You can check on the status of running jobs using `squeue`.

```
shell-prompt: squeue

<table>
<thead>
<tr>
<th>JOBID</th>
<th>PARTITION</th>
<th>NAME</th>
<th>USER</th>
<th>ST</th>
<th>TIME</th>
<th>NODES</th>
<th>NODELIST(REASON)</th>
</tr>
</thead>
<tbody>
<tr>
<td>64</td>
<td>default-p</td>
<td>bacon</td>
<td>R</td>
<td>0:02</td>
<td>2</td>
<td></td>
<td>compute-[001-002]</td>
</tr>
</tbody>
</table>
```

The Job id column shows the numeric job ID. The ST column shows the current status of the job. The most common status flags are 'PD' for pending (waiting to start) and 'R' for running.

The `squeue --long` flag requests more detailed information.

The `squeue` command has many flags for controlling what it reports. Run `man squeue` for full details.

As a convenience for users, our clusters have a script called `slurm-cluster-load`, which uses `squeue` and `sinfo` to display a quick summary on current jobs and overall load:

```
shell-prompt: slurm-cluster-load

<table>
<thead>
<tr>
<th>JOBID</th>
<th>USER</th>
<th>EXEC_HOST</th>
<th>CPUS</th>
<th>NODES</th>
<th>SHARED</th>
<th>NAME</th>
<th>TIME</th>
<th>ST</th>
</tr>
</thead>
<tbody>
<tr>
<td>186110</td>
<td>chen59</td>
<td>R</td>
<td>16</td>
<td>2</td>
<td>no surf</td>
<td>surf</td>
<td>1-08:46:19</td>
<td>compute-2-14,16</td>
</tr>
<tr>
<td>186112</td>
<td>chen59</td>
<td>R</td>
<td>16</td>
<td>2</td>
<td>no surf</td>
<td>surf</td>
<td>1-08:46:19</td>
<td>compute-2,compute-3-02</td>
</tr>
<tr>
<td>186113</td>
<td>chen59</td>
<td>R</td>
<td>16</td>
<td>2</td>
<td>no surf</td>
<td>surf</td>
<td>1-08:46:19</td>
<td>compute-3-07,10</td>
</tr>
<tr>
<td>187146</td>
<td>albertof</td>
<td>R</td>
<td>1</td>
<td>1</td>
<td>yes</td>
<td>bash</td>
<td>1-03:08:12</td>
<td>compute-2-04</td>
</tr>
<tr>
<td>187639</td>
<td>albertof</td>
<td>R</td>
<td>1</td>
<td>1</td>
<td>yes</td>
<td>submitSR</td>
<td>52:03</td>
<td>compute-4-02</td>
</tr>
<tr>
<td>187640</td>
<td>albertof</td>
<td>R</td>
<td>1</td>
<td>1</td>
<td>yes</td>
<td>submitSR</td>
<td>52:03</td>
<td>compute-4-02</td>
</tr>
<tr>
<td>187642</td>
<td>albertof</td>
<td>R</td>
<td>1</td>
<td>1</td>
<td>yes</td>
<td>submitSR</td>
<td>52:03</td>
<td>compute-4-02</td>
</tr>
<tr>
<td>187867</td>
<td>qium</td>
<td>R</td>
<td>8</td>
<td>1</td>
<td>no ph/IC1x1</td>
<td>46:11</td>
<td>compute-1-03</td>
<td></td>
</tr>
<tr>
<td>187868</td>
<td>qium</td>
<td>R</td>
<td>8</td>
<td>1</td>
<td>no ph/IC1x2</td>
<td>46:11</td>
<td>compute-1-04</td>
<td></td>
</tr>
</tbody>
</table>

CPUS(A/I/O/T)
785/264/87/1136
Load: 69%
```

12.1.3 Using top

Using the output from `squeue`, we can see which compute nodes are being used by a job:

```
shell-prompt: squeue

<table>
<thead>
<tr>
<th>JOBID</th>
<th>PARTITION</th>
<th>NAME</th>
<th>USER</th>
<th>TIME</th>
<th>NODES</th>
<th>NODELIST(REASON)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1017</td>
<td>batch</td>
<td>bash</td>
<td>oleary</td>
<td>2:44:00</td>
<td>1</td>
<td>compute-001</td>
</tr>
<tr>
<td>1031</td>
<td>batch</td>
<td>ARM97-pa</td>
<td>roberts</td>
<td>1:50:55</td>
<td>2</td>
<td>compute-(001-002)</td>
</tr>
<tr>
<td>1032</td>
<td>batch</td>
<td>sbatch</td>
<td>joea</td>
<td>1:35:19</td>
<td>1</td>
<td>compute-002</td>
</tr>
<tr>
<td>1034</td>
<td>batch</td>
<td>sbatch</td>
<td>joea</td>
<td>1:05:59</td>
<td>1</td>
<td>compute-002</td>
</tr>
<tr>
<td>1035</td>
<td>batch</td>
<td>sbatch</td>
<td>joea</td>
<td>50:08</td>
<td>1</td>
<td>compute-003</td>
</tr>
<tr>
<td>1036</td>
<td>batch</td>
<td>sbatch</td>
<td>joea</td>
<td>47:13</td>
<td>1</td>
<td>compute-004</td>
</tr>
<tr>
<td>1041</td>
<td>batch</td>
<td>bash</td>
<td>oleary</td>
<td>36:41</td>
<td>1</td>
<td>compute-001</td>
</tr>
<tr>
<td>1042</td>
<td>batch</td>
<td>bash</td>
<td>roberts</td>
<td>0:09</td>
<td>1</td>
<td>compute-001</td>
</tr>
</tbody>
</table>
```
From the above output, we can see that job 1031 is running on compute-001 through compute-002.

We can then examine the processes on any of those nodes using a remotely executed `top` command:

```
shell-prompt: ssh -t compute-001 top
```

```
top - 13:55:55 up 12 days, 24 min, 1 user, load average: 5.00, 5.00, 4.96
Tasks: 248 total, 6 running, 242 sleeping, 0 stopped, 0 zombie
Cpu(s): 62.5%us, 0.1%sy, 0.0%ni, 37.3%id, 0.1%wa, 0.0%hi, 0.0%si, 0.0%st
Mem: 24594804k total, 15388204k used, 9206600k free, 104540k buffers
Swap: 33554424k total, 10640k used, 33543784k free, 14486568k cached
```

<table>
<thead>
<tr>
<th>PID</th>
<th>USER</th>
<th>PR</th>
<th>NI</th>
<th>VIRT</th>
<th>RES</th>
<th>SHR</th>
<th>S</th>
<th>%CPU</th>
<th>%MEM</th>
<th>TIME+</th>
<th>COMMAND</th>
</tr>
</thead>
<tbody>
<tr>
<td>30144</td>
<td>roberts</td>
<td>20</td>
<td>0</td>
<td>251m</td>
<td>69m</td>
<td>17m</td>
<td>R</td>
<td>100.4</td>
<td>0.3</td>
<td>113:29.04</td>
<td>SAMADVMPDATA_1</td>
</tr>
<tr>
<td>30145</td>
<td>roberts</td>
<td>20</td>
<td>0</td>
<td>251m</td>
<td>69m</td>
<td>17m</td>
<td>R</td>
<td>100.4</td>
<td>0.3</td>
<td>113:28.16</td>
<td>SAMADVMPDATA_1</td>
</tr>
<tr>
<td>30143</td>
<td>roberts</td>
<td>20</td>
<td>0</td>
<td>251m</td>
<td>69m</td>
<td>17m</td>
<td>R</td>
<td>100.1</td>
<td>0.3</td>
<td>115:16.87</td>
<td>SAMADVMPDATA_1</td>
</tr>
<tr>
<td>30146</td>
<td>roberts</td>
<td>20</td>
<td>0</td>
<td>251m</td>
<td>69m</td>
<td>17m</td>
<td>R</td>
<td>100.1</td>
<td>0.3</td>
<td>113:29.24</td>
<td>SAMADVMPDATA_1</td>
</tr>
<tr>
<td>30147</td>
<td>roberts</td>
<td>20</td>
<td>0</td>
<td>251m</td>
<td>69m</td>
<td>17m</td>
<td>R</td>
<td>100.1</td>
<td>0.3</td>
<td>113:29.82</td>
<td>SAMADVMPDATA_1</td>
</tr>
<tr>
<td>965</td>
<td>bacon</td>
<td>20</td>
<td>0</td>
<td>15168</td>
<td>1352</td>
<td>944</td>
<td>R</td>
<td>0.3</td>
<td>0.0</td>
<td>0:00.01</td>
<td>top</td>
</tr>
<tr>
<td>1</td>
<td>root</td>
<td>20</td>
<td>0</td>
<td>19356</td>
<td>1220</td>
<td>1004</td>
<td>S</td>
<td>0.0</td>
<td>0.0</td>
<td>0:00.96</td>
<td>init</td>
</tr>
</tbody>
</table>

The column of interest is under "RES".

We can see from the `top` command above that the processes owned by job 1031 are using about 69 mebibytes of memory each. Watch this value for a while as it will change as the job runs.

Take the highest value you see in the RES column, add about 10%, and use this with --mem-per-cpu to set a reasonable memory limit.

```
SBATCH --mem-per-cpu=76
```

A convenience script is provided on UWM clusters to save a little typing when running `top`:

```
shell-prompt: topnode 001
```

### 12.1.4 Job Submission

The purpose of this section is to provide the reader a quick start in job scheduling using the most common tools. The full details of job submission are beyond the scope of this document. For more information, see SLURM website [http://slurm.schedmd.com/](http://slurm.schedmd.com/) and the man pages for individual SLURM commands.

```
man sbatch
```

#### Submission Scripts

Submitting jobs involves specifying a number of job parameters such as the number of cores, the job name (which is used by other SLURM commands), the name(s) of the output file(s), etc.

In order to document all of this information and make it easy to resubmit the same job, this information is usually incorporated into a *submission script*. Using a script saves you a lot of typing when you want to run-submit the same job, and also fully documents the job parameters.

A submission script is an ordinary shell script, with some *directives* inserted to provide information to the scheduler. For SLURM, the directives are specially formatted shell comments beginning with "#SBATCH".
Note
The ONLY difference between an sbatch submission script and a script that you would run on any Unix laptop or workstation is the #SBATCH directives.
You can develop and test a script to run your analyses or models on any Unix system. To use it on a SLURM cluster, you need only add the appropriate #SBATCH directives and possibly alter some command arguments to enable parallel execution.

Caution
There cannot be any Unix commands above #SBATCH directives. SLURM will ignore any #SBATCH directives below the first Unix command.

Submission scripts are submitted with the sbatch command. The script may contain #SBATCH options to define the job, regular Unix commands, and srun or other commands to run programs in parallel.

Note
The script submitted by sbatch is executed on one core, regardless of how many cores are allocated for the job. The commands within the submission script are responsible for dispatching multiple processes for parallel jobs. This strategy differs from other popular schedulers like Torque (PBS) and LSF, where the submission script is run in parallel on all cores.

Suppose we have the following text in a file called hostname.sbatch:

```bash
#!/usr/bin/env bash
# A SLURM directive
#SBATCH --job-name=hostname

# A command to be executed on the scheduled node.
# Prints the host name of the node running this script.
hostname
```

The script is submitted to the SLURM scheduler as a command line argument to the sbatch command:

```
shell-prompt: sbatch hostname.sbatch
```

The SLURM scheduler finds a free core on a compute node, reserves it, and then remotely runs hostname.sbatch on the compute node using ssh or some other remote execution command.

Recall from Chapter 8 that everything in a shell script from a `#` character to the end of the line is considered a comment by the shell, and ignored.

However, comments beginning with #SBATCH, while ignored by the shell, are interpreted as directives by sbatch.

The directives within the script provide command line flags to sbatch. For instance, the line

```
#SBATCH --mem-per-cpu=10
```

causes sbatch to behave as if you had typed

```
shell-prompt: sbatch --mem-per-cpu=10 hostname.sbatch
```

By putting these comments in the script, you eliminate the need to remember them and retype them every time you run the job. It’s generally best to put all sbatch flags in the script rather than type any of them on the command line, so that you have an exact record of how the job was started. This will help you determine what went wrong if there are problems, and allow you to reproduce the results at a later date.

Note
If you want to disable a #SBATCH comment, you can just add another `#` rather than delete it. This will allow you to easily enable it again later as well as maintain a record of options you used previously.
Practice Break

```bash
#!/usr/bin/env bash

# A SLURM directive
#SBATCH --job-name=hostname

# A command to be executed on the scheduled node.
# Prints the host name of the node running this script.
hostname
```

Type in the `hostname.sbatch` script shown above and submit it to the scheduler using `sbatch`. Then check the status with `squeue` and view the output and error files.

Definitions

A **CPU or core** is a processor capable of running a program.

A **node** is a distinct computer within a cluster. It has one or more CPUs and a memory space that is separate from other nodes.

A **process** is a process as defined by Unix. (The execution of a program.) A parallel job on a cluster consists of multiple processes running at the same time. A serial job consists of only one process.

A **job** is all of the processes dispatched by SLURM under the same job ID. Processes within a job on a cluster may be running on the same node or different nodes.

A **task** is one or more processes within a job that are distinct from the rest of the processes. All processes within a given task must run on the same node. For example, a job could consist of 4 tasks running on different nodes, each consisting of 8 threads using shared-memory parallelism. This job therefore uses 32 cores and a maximum of 4 different nodes. Two tasks could run on the same node if the node has at least 16 cores.

Common Flags

```bash
#SBATCH --output=standard-output-file
#SBATCH --error=standard-error-file
#SBATCH --nodes=min[-max]
#SBATCH --ntasks=tasks
#SBATCH --array=list
#SBATCH --cpus-per-task=N
#SBATCH --exclusive
#SBATCH --mail-type=BEGIN|END|FAIL|REQUEUE|ALL
#SBATCH --mail-user=email-address
#SBATCH --mem=MB
#SBATCH --mem-per-cpu=MB
#SBATCH --partition=name
```

The `--output` and `--error` flags control the names of the files to which the standard output and standard error of the processes are redirected. If omitted, both standard output and standard error are written to `slurm-JOBID.out`.

*Note* Commands in a submission script should not use output redirection (> or error redirection (2>), >&) since these would conflict with `--output` and `--error`. 
The \texttt{--ntasks} flag indicates how many tasks the job will run, usually for multicore jobs. The \texttt{--nodes} flag is used to specify how many nodes (not tasks) to allocate for the job. We can use this to control how many tasks are run on each node. For example, if a job consists of 20 I/O-intensive processes, we would not want many of them running on the same node and competing for the local disk. In this case, we can specify \texttt{--nodes=20 \--ntasks=20} to force each process to a separate node.

\textbf{Note} The \texttt{sbatch} command with \texttt{--nodes} or \texttt{--ntasks} will not cause multiple processes to run. The \texttt{sbatch} command simply runs the script on one node. To run multiple tasks, you must use \texttt{srun}, \texttt{mpirun}, or some other dispatcher within the script. This differs from many other schedulers such as LSF and Torque.

The \texttt{--cpus-per-task=N} flag indicates that we need \(N\) cpus on the same node. Using \texttt{--ntasks=4 \--cpus-per-task=3} will indicate 4 tasks using 3 cores each, in effect requesting 12 cores in groups of 3 per node.

The \texttt{--exclusive} flag indicates that the job cannot share nodes with other jobs. This is typically used for shared memory parallel jobs to maximize the number of cores available to the job. It may also be used for jobs with high memory requirements, although it is better to simply specify the memory requirements using \texttt{--mem} or \texttt{--mem-per-cpu}.

The \texttt{--mail-type} flag indicates when email should be sent, e.g. when the job begins, ends, fails, is requeued, or all of the above. The \texttt{--mail-user} flag indicates the email address to which notifications are sent.

The \texttt{--mem=MB} flag indicates the amount of memory needed on each node used by the job, in megabytes.

The \texttt{--mem-per-cpu=MB} flag indicates the amount of memory needed by each process within a job, in megabytes.

The \texttt{--partition=name} flag indicates which partition (set of nodes) on which the job should run. Simply run \texttt{sinfo} to see a list of available partitions.

\textbf{Note} UWM users should be aware of the "-nice" partitions. These partitions run jobs at a lower priority, allowing them to be preempted by other jobs if necessary. If you want to use more than your fair share of cores, simply submit your fair share to normal partitions such as "batch", and submit additional jobs to a partition such as "batch-nice". This way, you can use as many cores as you want while they are available, but if other users need them, your jobs in "batch-nice" will be terminated automatically.

\subsection*{SLURM Resource Requirements}

When using a cluster, it is important to develop a feel for the resources required by your jobs, and inform the scheduler as accurately as possible what will be needed in terms of CPU time, memory, etc.

If a user does not specify a given resource requirement, the scheduler uses default limits. Default limits are set low, so that users are encouraged to provide an estimate of required resources for all non-trivial jobs. This protects other users from being blocked by long-running jobs that require less memory and other resources than the scheduler would assume.

The \texttt{--mem=MB} flag indicates that the job requires \(MB\) megabytes of memory per node.

The \texttt{--mem-per-cpu=MB} flag indicates that the job requires \(MB\) megabytes per core.

\textbf{Note} It is very important to specify memory requirements accurately in all jobs, and it is generally easy to predict based on previous runs by monitoring processes within the job using \texttt{top} or \texttt{ps}. Failure to do so could block other jobs running, even though the resources it requires are actually available.

\subsection*{Batch Serial Jobs}

A batch serial submission script need only have optional flags such as job name, output file, etc. and one or more commands.

\begin{verbatim}
#!/usr/bin/env bash
hostname
\end{verbatim}
Interactive Jobs

Interactive jobs involve running programs under the scheduler on compute nodes, where the user can interact directly with the process. I.e., output is displayed on their terminal instead of being redirected to files (as controlled by the `sbatch --output` and `--error` flags) and input can be taken from the keyboard.

The use of interactive jobs on a cluster is uncommon, but sometimes useful.

For example, an interactive shell environment might be useful when porting a new program to the cluster. This can be used to do basic compilation and testing in the compute node environment before submitting larger jobs. Programs should not be compiled or tested on the head node and users should not ssh directly to a compute node for this purpose, as this would circumvent the scheduler, causing their compilations, etc. to overload the node.

---

Caution A cluster is not a good place to do code development and testing. Code should be fully developed and tested on a separate development systems such as a laptop or workstation before being run on a cluster, where buggy programs will waste valuable shared resources and may cause other problems for other users.

---

On UWM clusters, a convenience script called `slurm-shell` is provided to make it easy to start an interactive shell under the scheduler.

```bash
shell-prompt: slurm-shell
```

Note:

`slurm-shell` is provided solely as a convenience for typical users.

If you would like an interactive shell with additional memory, more than 1 CPU, etc., you can use `salloc` and `srun` directly. Run

```bash
more /usr/local/bin/slurm-shell
```

for a basic example and

```bash
man salloc
man srun
```

for full details.

---

1:05PM up 22 days, 2 hrs, 0 users, load averages: 0.00, 0.00, 0.00
To repeat the last command in the C shell, type "!!".
-- Dru <genesis@istar.ca>
FreeBSD compute-001. bacon ~ 401:

Batch Parallel Jobs (Job Arrays)

A job array is a set of independent processes all started by a single job submission. The entire job array can be treated as a single job by SLURM commands such as `squeue` and `scancel`.

A batch parallel submission script looks almost exactly like a batch serial script. There are a couple of ways to run a batch parallel job. It is possible to run a job array using `--ntasks` and an `srun` command in the `sbatch` script. However, the `--array` flag is far more convenient for most purposes.

The `--array` flag is followed by an index specification which allows the user to specify any set of task IDs for each job in the array. The specification can use `-` to specify a range of task IDs, commas to specify an arbitrary list, or both. For example, to run a job array with task IDs ranging from 1 to 5, we would use the following:
Note
All UWM clusters use an operating system feature known as task affinity to improve performance. The feature binds each process to a specific core, so that any data in the cache RAM of that core does not have to be flushed and reloaded into the cache of a different core. This can dramatically improve performance for memory-intensive processes.
In order to activate task affinity, you must use `srun` or another command to launch the processes.

```bash
#!/usr/bin/env bash
#SBATCH --array=1-5
# OK, but will not use task affinity
hostname
```

```bash
#!/usr/bin/env bash
#SBATCH --array=1-5
# Activates task affinity, so each hostname process will stick to one core
srun hostname
```

If we wanted task IDs of 1,2,3,10, 11, and 12, we could use the following:

```bash
#!/usr/bin/env bash
#SBATCH --array=1-3,10-12
srun hostname
```

The reasons for selecting specific task IDs are discussed in Section 12.1.4.

Note Each task in an array job is treated as a separate job by slurm. Hence, `--mem` and `--mem-per-cpu` mean the same thing in an array job. Both refer to the memory required by one task.

Another advantage of using `--array` is that we don’t need all the cores available at once in order for the job to run. For example, if we submit a job using `--array=1-200` while there are only 50 cores free on the cluster, it will begin running as many processes as possible immediately and run more as more cores become free. In contrast, a job started with `--ntasks` will remain in a pending state until there are 200 cores free.

Furthermore, we can explicitly limit the number of cores we use at once with a simple addition to the flag. Suppose we want to run 10,000 processes, but be nice to other cluster users and only use 100 cores at a time. All we need to do is use the following:

```bash
#SBATCH --array=1-10000%100
```

Caution Do not confuse this with `--array=1-1000:4`, which simply increments the job index by 4 instead of one, and will attempt to run 10,000 processes at once using indices 1, 5, 9, ...!

Practice Break
Copy your `hostname.sbatch` to `hostname-parallel.sbatch`, modify it to run 5 processes, and submit it to the scheduler using `sbatch`. Then check the status with `squeue` and view the output and error files.
**Multi-core Jobs**

A multi-core job is any job that runs a parallel program. This means that it uses multiple processes that communicate and cooperate with each other in some way. There is no such communication or cooperation in batch parallel (also known as embarrassingly parallel) jobs, which use SLURM’s --array flag.

Cores are allocated for multi-core jobs using the --ntasks flag.

Using --ntasks ONLY tells the SLURM scheduler how many cores to reserve. It does NOT tell your parallel program how many cores or which compute nodes to use. It is the responsibility of the user to ensure that the command(s) in their script utilize the correct number of cores and the correct nodes within the cluster.

Some systems, such as OpenMPI (discussed in Section 36.8), will automatically detect this information from the SLURM environment and dispatch the correct number of processes to each allocated node.

If you are not using OpenMPI, you may need to specify the number of cores and/or which nodes to use in the command(s) that run your computation. Many commands have a flag argument such as -n or -np for this purpose.

OpenMP (not to be confused with OpenMPI) is often used to run multiple threads on the same node (shared memory parallelism, as discussed in Section 10.2 and Section 36.6). OpenMP software will look for the OMP_NUM_THREADS environment variable to indicate how many cores to use. It is the user’s responsibility to ensure that OMP_NUM_THREADS is set when using OpenMP software.

**MPI Multi-core Jobs**

Scheduling MPI jobs in SLURM is much like scheduling batch parallel jobs.

MPI programs cannot be executed directly from the command line as we do with normal programs and scripts. Instead, we must use the **mpirun** or **srun** command to start up MPI programs.

⚠️ **Caution** Like any other command used on a cluster or grid, **mpirun** must not be executed directly from the command line, but instead must be used in a scheduler submission script.

For MPI and other multicore jobs, we use the sbatch --ntasks flag to indicate how many cores we want to use. Unlike --array, sbatch with --ntasks runs the sbatch script on only one node, and it is up to the commands in the script (such as mpirun or srun) to dispatch the parallel processes to all the allocated cores.

Commands like mpirun and srun can retrieve information about which cores have been allocated from the environment handed down by their parent process, the SLURM scheduler. The details about SLURM environment variables are discussed in Section 12.1.4.

```bash
#!/bin/sh
#SBATCH --ntasks=2
PATH=${PATH}:/usr/local/mpi/openmpi/bin
export PATH
mpirun ./mpi_bench
```
When running MPI jobs, it is often desirable to have as many processes as possible running on the same node. Message passing is generally faster between processes on the same node than between processes on different nodes, because messages passed within the same node need not cross the network. If you have a very fast network such as Infiniband or a low-latency Ethernet, the difference may be marginal, but on more ordinary networks such as gigabit Ethernet, the difference can be enormous.

SLURM by default will place as many processes as possible on the same node. We can also use --exclusive to ensure that "leftover" cores on a partially busy node will not be used for our MPI jobs. This may improve communication performance, but may also delay the start of the job until enough nodes are completely empty.

Environment Variables

SLURM sets a number of environment variables when a job is started. These variables can be used in the submission script and within other scripts or programs executed as part of the job.

```
shell-prompt: srun printenv | grep SLURM
SLURM_SRUN_COMM_PORT=27253
SLURM_TASKS_PER_NODE=1
SLURM_NODELIST=compute-001
SLURM_JOB_NUM_NODES=1
SLURM_NNODES=1
SLURM_STEPID=0
SLURM_STEP_ID=0
SLURM_JOBID=81
SLURM_JOB_ID=81
SLURM_DISTRIBUTION=cyclic
SLURM_NPROCS=1
SLURM_NTASKS=1
SLURM_JOB_CPUS_PER_NODE=1
SLURM_JOB_NAME=/usr/bin/printenv
SLURM_SUBMIT_HOST=finch.cs.uwm.edu
SLURM_SUBMIT_DIR=/usr/home/bacon
SLURM_PRIO_PROCESS=0
SLURM_STEP_NODELIST=compute-001
SLURM_STEP_ID=0
SLURM_STEP_TASKS_PER_NODE=1
SLURM_STEP_LAUNCHER_PORT=27253
SLURM_SRUN_COMM_HOST=192.168.0.2
SLURM_STEP_TOPOLOGY_ADDR=compute-001
SLURM_TOPOLOGY_ADDR_PATTERN=node
SLURM_CPUS_ON_NODE=1
SLURM_TASK_PID=5945
SLURM_NODEID=0
SLURM_PROCID=0
SLURM_LOCALID=0
SLURM_LAUNCH_NODE_IPADDR=192.168.0.2
SLURM_GTIDS=0
SLURM_CHECKPOINT_IMAGE_DIR=/usr/home/bacon
SLURM_NODENAME=compute-001
```

The SLURM_JOB_NAME variable can be useful for generating output file names within a program, among other things.

When submitting a job array using --array, the submission script is dispatched to every core by sbatch. To distinguish between tasks in this type of job array, we would examine SLURM_ARRAY_TASK_ID. This variable will be set to a different value for each task, from the specification given with --array.

For example, suppose we have 100 input files named input-1.txt through input-100.txt. We could use the following script to process them:

```
#!/usr/bin/env bash
#SBATCH --array=1-100
```
Suppose our input files are not numbered sequentially, but according to some other criteria, such as a set of prime numbers. In this case, we would simply change the specification in --array:

```bash
#!/usr/bin/env bash
#SBATCH --array=2,3,5,7,11,13
../myprog < input-$SLURM_ARRAY_TASK_ID.txt > output-$SLURM_ARRAY_TASK_ID.txt
```

12.1.5 Terminating a Job

If you determine that a job is not behaving properly (by reviewing partial output, for example), you can terminate it using `scancel`, which takes a job ID as a command line argument.

```
shell-prompt: sbatch bench.sbatch
Submitted batch job 90
shell-prompt: squeue
JOBID PARTITION NAME USER ST TIME NODES NODELIST(REASON)
 90 default-p bench.sl bacon R 0:03 2 compute-[001-002]
shell-prompt: scancel 90
shell-prompt: squeue
JOBID PARTITION NAME USER ST TIME NODES NODELIST(REASON)
```

12.1.6 Terminating Stray Processes

Occasionally, a SLURM job may fail and leave processes running on the compute nodes. These are called stray processes, since they are no longer under the control of the scheduler.

Do not confuse stray processes with zombie processes. A zombie process is a process that has terminated but has not been reaped. I.e., it is still listed by the `ps` command because its parent process has not become aware that it is terminated. Zombie processes are finished and do not consume any resources, so we need not worry about them.

Stray processes are easy to detect on nodes where you have no jobs running. If `squeue` or `slurm-cluster-load` do not show any of your jobs using a particular node, but `top` or `topnode` show processes under your name, then you have some strays. Simply `ssh` to that compute node and terminate them with the standard Unix `kill` or `killall` commands.

If you have jobs running on the same node as your strays, then detecting the strays may be difficult. If the strays are running a different program than your legitimate job processes, then they will be easy to spot. If they are running the same program as your legitimate job processes, then they will likely have a different run time than your legitimate processes. Be very careful to ensure that you kill the strays and not your active job in this situation.

```
Linux login.mortimer bacon ~ 414: squeue -u bacon
JOBID PARTITION NAME USER ST TIME NODES NODELIST(REASON)
Linux login.mortimer bacon ~ 415: topnode 003
top - 09:33:55 up 2 days, 10:44, 1 user, load average: 16.02, 15.99, 14.58
Tasks: 510 total, 17 running, 493 sleeping, 0 stopped, 0 zombie
Cpu(s): 2.3%us, 0.0%sy, 0.0%ni, 97.7%id, 0.0%wa, 0.0%hi, 0.0%si, 0.0%st
Mem: 49508900k total, 2770060k used, 46738840k free, 62868k buffers
Swap: 33554428k total, 0k used, 33554428k free, 278916k cached
PID USER PR NI VIRT RES SHR S %CPU %MEM TIME+ COMMAND
7530 bacon 20 0 468m 112m 4376 R 81.5 0.2 37:55.11 mpi-bench
7531 bacon 20 0 468m 112m 4524 R 81.5 0.2 37:55.44 mpi-bench
```
If the normal kill command doesn’t work, you can use `kill -9` to do a sure kill.

12.1.7 Viewing Output of Active Jobs

Unlike many other schedulers, SLURM output files are available for viewing while the job is running. Hence, SLURM does not require a “peek” command. The default output file or files specified by `--output` and `--error` are updated regularly while a job is running and can be viewed with standard Unix commands such as “more”.

12.1.8 Checking Job Stats with `sacct`

The `sacct` command is used to view accounting statistics on completed jobs. With no command-line arguments, `sacct` prints a summary of your past jobs:

```
shell-prompt: sacct
          JobID  JobName   Partition  Account AllocCPUS State  ExitCode
------------ ---------- ---------- ---------- ---------- -------- --------
   70  build.slu+ default-p+ (null)      1  COMPLETED 0:0
   71  build.slu+ default-p+ (null)      1  COMPLETED 0:0
   72  bench.slu+ default-p+ (null)     24  COMPLETED 0:0
...
   184 hostname default-p+ (null)      40  COMPLETED 0:0
  184.0 hostname (null)              40  COMPLETED 0:0
   185 bench-fre+ default-p+ (null)     48  CANCELLED 0:0
  185.0 orted (null)                3  FAILED 1:0
   186 env.sbatch default-p+ (null)      1  COMPLETED 0:0
```

For detailed information on a particular job, we can use the `-j` flag to specify a job id and the `-o` flag to specify which information to display:

```
shell-prompt: sacct -o alloccpus,nodeList,elapsed,cputime -j 117
          AllocCPUS  NodeList  Elapsed  CPUTime
-------------- -------------- ---------- ----------
         12 compute-001 00:00:29 00:05:48
```
On UWM SLURM clusters, the above command is provided in a convenience script called slurm-job-stats:

```
shell-prompt: slurm-job-stats 117
AllocCPUS NodeList Elapsed CPUTime
---------- --------------- ---------- ----------
12 compute-001 00:00:29 00:05:48
```

For more information on sacct, run "man sacct" or view the online SLURM documentation.

### 12.1.9 Checking Status of Running Jobs with scontrol

Detailed information on currently running jobs can be displayed using scontrol.

```
shell-prompt: scontrol show jobid 10209
JobId=10209 JobName=bench-freebsd.sbatch
  UserId=bacon(4000) GroupId=bacon(4000)
  Priority=4294901512 Nice=0 Account=(null) QOS=(null)
  JobState=FAILED Reason=NonZeroExitCode Dependency=(null)
  RunTime=00:00:02 TimeLimit=UNLIMITED TimeMin=N/A
  PreemptTime=None SuspendTime=None SecsPreSuspend=0
  Partition=default-partition AllocNode:Sid=login:7345
  ReqNodeList=(null) ExcNodeList=(null)
  NodeList=compute-[003-006]
  BatchHost=compute-003
  NumNodes=4 NumCPUs=48 CPUs/Task=1 ReqB:S:C:T=0:0:*:* CoreSpec=* Socks/Node=* NtasksPerNode:B:S:C=0:0:*:* CoreSpec=* MinCPUsNode=1 MinMemoryCPU=1024M MinTmpDiskNode=0 Features=(null) Gres=(null) Reservation=(null) Command=/share1/Data/bacon/Facil/Software/Src/Bench/MPI/bench-freebsd.sbatch
  WorkDir=/share1/Data/bacon/Facil/Software/Src/Bench/MPI
  StdErr=/share1/Data/bacon/Facil/Software/Src/Bench/MPI/slurm-10209.out
  StdIn=/dev/null
  StdOut=/share1/Data/bacon/Facil/Software/Src/Bench/MPI/slurm-10209.out
```

On UWM SLURM clusters, the above command is provided in a convenience script called slurm-job-status:

```
shell-prompt: squeue
  JOBID PARTITION NAME USER ST  TIME  NODES NODELIST(REASON)
  20537 batch bench.sb bacon R 0:02 1 compute-001
```

```
shell-prompt: slurm-job-status 20537
JobId=20537 JobName=bench.sbatch
  UserId=bacon(4000) GroupId=bacon(4000)
  Priority=4294900736 Nice=0 Account=(null) QOS=(null)
  JobState=RUNNING Reason=None Dependency=(null)
  RunTime=00:00:12 TimeLimit=UNLIMITED TimeMin=N/A
  StartTime=2016-07-18T11:38:31 EndTime=Unknown
  PreemptTime=None SuspendTime=None SecsPreSuspend=0
  Partition=batch AllocNode:Sid=login:7485
  ReqNodeList=(null) ExcNodeList=(null)
  NodeList=compute-001
  BatchHost=compute-001
  NumNodes=1 NumCPUs=12 CPUs/Task=1 ReqB:S:C:T=0:0:*:* CoreSpec=*
  TRES=cpu=12,mem=3072,node=1
  Socks/Node=* NtasksPerNode:B:S:C=0:0:*:* CoreSpec=*
12.1.10 Job Sequences

If you need to submit a series of jobs in sequence, where one job begins after another has completed, the simplest approach is to simply submit job N+1 from the sbatch script for job N.

It’s important to make sure that the current job completed successfully before submitting the next, to avoid wasting resources. It is up to you to determine the best way to verify that a job was successful. Examples might include grepping the log file for some string indicating success, or making the job create a marker file using the `touch` command after a successful run. If the command used in your job returns a Unix-style exit status (0 for success, non-zero on error), then you can simply use the shell’s exit-on-error feature to make your script exit when any command fails. Below is a template for scripts that might run a series of jobs.

```sh
#!/bin/sh

#SBATCH job-parameters
set -e  # Set exit-on-error

job-command

# This script will exit here if job-command failed

sbatch job2.sbatch  # Executed only if job-command succeeded
```

12.1.11 Self-test

1. What is the SLURM command for showing the current state of all nodes in a cluster?

2. What is the SLURM command to show the currently running jobs on a cluster?

3. Write and submit a batch-serial SLURM script called `list-etc.sbatch` that prints the host name of the compute node on which it runs and a long-listing of `/etc` directory on that node.
   The script should store the output of the commands in `list-etc.stdout` and error messages in `list-etc.stderr` in the directory from which the script was submitted.
   The job should appear in `squeue` listings under the name "list-etc". Quickly check the status of your job after submitting it.

4. Copy your `list-etc.sbatch` script to `list-etc-parallel.sbatch`, and modify it so that it runs the `hostname` and `ls` commands on 10 cores instead of just one.
   The job should produce a separate output file for each process named `list-etc-parallel.o<jobid>-<arrayid>` and a separate error file for each process named `list-etc-parallel.e<jobid>-<arrayid>`.
   Quickly check the status of your job after submitting it.

5. What is the SLURM command for terminating a job with job-id 3545?

6. What is the SLURM command for viewing the terminal output of a job with job-id 3545 while it is still running?

7. What is the SLURM command for showing detailed job information about the job with job-id 3254?
12.2 Local Customizations

TBD
Chapter 13

Job Scheduling with HTCondor

Before You Begin

Before reading this chapter, you should be familiar with basic Unix concepts (Chapter 7), the Unix shell (Section 7.4.3, redirection (Section 7.17.1), shell scripting (Chapter 8), and job scheduling (Chapter 11).

For complete information, see the official HTCondor documentation at http://research.cs.wisc.edu/htcondor/.

13.1 The HTCondor Resource Manager

The goal of this section is to provide a quick introduction to the HTCondor scheduler. For more detailed information, consult the HTCondor man pages and the HTCondor website.

13.1.1 A Unique Approach to Resource Management

HTCondor is very different from other common resource managers because it serves different goals.

HTCondor is intended for use on "distributively owned" resources, i.e. PCs that are owned by people other than the HTCondor users and used primarily for purposes other than parallel computing. Most other resource managers are designed for use on dedicated resources, such as a cluster of computers built specifically for parallel computation.

As such, HTCondor is designed to "borrow" resources from their owners while the owners are not otherwise using them. The PC owner has the right to "evict" (terminate) an HTCondor process at any time and without warning. In fact, if an owner logs into a PC while an HTCondor process is running, the process is generally evicted and restarted elsewhere.

HTCondor can be configured so that processes are merely suspended or even allowed to continue running, but this requires permission from the owner of the PC. Since the PCs on an HTCondor pool may be owned by many different people, it’s best to assume that at least some of the PCs are configured to evict HTCondor processes when a local user is active.

A computer used by HTCondor should therefore not be expected to remain available for long periods of time. Plan for some of the processes running under HTCondor to be terminated and restarted on a different machine.

The longer an HTCondor process runs, the less likely it is to complete. As a rule of thumb, an HTCondor process should ideally run no more than a few hours. This is usually easy to achieve. Most HTCondor users are running embarrassingly parallel jobs that simply divide up input data or parameter space among an arbitrary number of independent processes. If such a user has 1,000 hours worth of computation, they can just as easily do it with 10 jobs running for 100 hours each, 100 jobs running for 10 hours each, or 500 jobs running for 2 hours each.

Don’t make your processes too short, though. Each process entails some scheduling overhead, and may take up to a minute or so to start. We want our jobs to spend much more time running than sitting in a queue in order to maximize throughput. There is little benefit to breaking computation into jobs much shorter than an hour and scheduling overhead will become significant if you do.
One of the advantages of HTCondor is that the number of available cores is usually irrelevant to how you divide up your job. HTCondor will simply run as many processes as it can at any given moment and start others as cores become available. For example, even if you only have 200 cores available, you can run a job consisting of 500 processes. HTCondor will run up to 200 at a time until all 500 are finished. 500 processes running for 2 hours each is usually preferable to 100 processes of 10 hours each, since this reduces the risk of individual processes being evicted from a PC.

### 13.1.2 HTCondor Terminology

- **An execute host** is a computer available for use by HTCondor for running a user's processes.
- **A job** is usually one process running on an execute host managed by HTCondor.
- **A cluster** (not to be confused with a hardware cluster) is the group of jobs from a single condor_submit, all of which share the same numeric ID.
- **A submit host** is a computer from which HTCondor jobs can be submitted.
- **A central manager** is a computer that manages a specific pool of computers which include execute hosts, submit hosts, and possibly other types of hosts. Every HTCondor pool has its own central manager.
- **A grid** is a group of computers available for parallel computing. It consists of one or more pools, each with its own central manager.
- **Flocking** is a system that allows jobs from one pool that has run out of resources to migrate to another pool, much like a flock of birds migrating when they run out of food.

### 13.1.3 Pool Status

Before scheduling any jobs on through HTCondor, it is often useful to check the status of the hosts.

The `condor_status` command shows the status of all the execute hosts in the pool.

```
FreeBSD peregrine bacon ~ 402: condor_status | more

<table>
<thead>
<tr>
<th>Name</th>
<th>OpSys</th>
<th>Arch</th>
<th>State</th>
<th>Activity</th>
<th>LoadAv</th>
<th>Mem</th>
<th>ActvtyTime</th>
</tr>
</thead>
<tbody>
<tr>
<td>povbEEFF73C04D8.1</td>
<td>LINUX</td>
<td>INTEL</td>
<td>Backfill</td>
<td>Idle</td>
<td>0.050</td>
<td>1968</td>
<td>+00:00:03</td>
</tr>
<tr>
<td>povbEEFF73C04485.1</td>
<td>LINUX</td>
<td>INTEL</td>
<td>Backfill</td>
<td>Idle</td>
<td>0.080</td>
<td>1968</td>
<td>+00:00:04</td>
</tr>
<tr>
<td>povbEEFF73C046A1.1</td>
<td>LINUX</td>
<td>INTEL</td>
<td>Backfill</td>
<td>Idle</td>
<td>0.020</td>
<td>1968</td>
<td>+00:00:04</td>
</tr>
<tr>
<td>povbEEFF73C048DB.1</td>
<td>LINUX</td>
<td>INTEL</td>
<td>Backfill</td>
<td>Idle</td>
<td>0.000</td>
<td>1968</td>
<td>+00:00:04</td>
</tr>
<tr>
<td>povbEEFF73C04BF1.1</td>
<td>LINUX</td>
<td>INTEL</td>
<td>Backfill</td>
<td>Idle</td>
<td>0.130</td>
<td>1968</td>
<td>+00:00:04</td>
</tr>
<tr>
<td>povbEEFF73C0516E.1</td>
<td>LINUX</td>
<td>INTEL</td>
<td>Backfill</td>
<td>Idle</td>
<td>0.110</td>
<td>1968</td>
<td>+00:00:04</td>
</tr>
<tr>
<td>povbEEFF73C061C5.1</td>
<td>LINUX</td>
<td>INTEL</td>
<td>Backfill</td>
<td>Idle</td>
<td>0.050</td>
<td>1968</td>
<td>+00:00:04</td>
</tr>
<tr>
<td>povbEEFF73C06F85.1</td>
<td>LINUX</td>
<td>INTEL</td>
<td>Backfill</td>
<td>Idle</td>
<td>0.070</td>
<td>1968</td>
<td>+00:00:04</td>
</tr>
<tr>
<td>povbEEFF73C0873B.1</td>
<td>LINUX</td>
<td>INTEL</td>
<td>Backfill</td>
<td>Idle</td>
<td>0.130</td>
<td>1968</td>
<td>+00:00:04</td>
</tr>
<tr>
<td>povbEEFF73C08BA6.1</td>
<td>LINUX</td>
<td>INTEL</td>
<td>Backfill</td>
<td>Idle</td>
<td>0.150</td>
<td>1968</td>
<td>+00:00:04</td>
</tr>
<tr>
<td>povbEEFF73C08E57.1</td>
<td>LINUX</td>
<td>INTEL</td>
<td>Backfill</td>
<td>Idle</td>
<td>0.020</td>
<td>1968</td>
<td>+00:00:04</td>
</tr>
<tr>
<td>povbEEFF73C08F4P.1</td>
<td>LINUX</td>
<td>INTEL</td>
<td>Backfill</td>
<td>Idle</td>
<td>0.070</td>
<td>1968</td>
<td>+00:00:04</td>
</tr>
<tr>
<td>povbEEFFA0A4F157.1</td>
<td>LINUX</td>
<td>INTEL</td>
<td>Owner</td>
<td>Idle</td>
<td>0.290</td>
<td>1009</td>
<td>+00:00:04</td>
</tr>
<tr>
<td>povbEEFFA0A6F42.1</td>
<td>LINUX</td>
<td>INTEL</td>
<td>Backfill</td>
<td>Idle</td>
<td>0.090</td>
<td>1009</td>
<td>+00:32:43</td>
</tr>
<tr>
<td>povbEEFFA0A717C.1</td>
<td>LINUX</td>
<td>INTEL</td>
<td>Backfill</td>
<td>Idle</td>
<td>0.370</td>
<td>1009</td>
<td>+00:00:04</td>
</tr>
<tr>
<td>povbEEFFA0D6A4C.1</td>
<td>LINUX</td>
<td>INTEL</td>
<td>Backfill</td>
<td>Idle</td>
<td>0.100</td>
<td>1009</td>
<td>+00:00:04</td>
</tr>
<tr>
<td>povbEEFFA0B8E990.1</td>
<td>LINUX</td>
<td>INTEL</td>
<td>Backfill</td>
<td>Idle</td>
<td>0.080</td>
<td>1009</td>
<td>+00:00:04</td>
</tr>
<tr>
<td>povbEEFFA0DC5C2.1</td>
<td>LINUX</td>
<td>INTEL</td>
<td>Backfill</td>
<td>Idle</td>
<td>0.140</td>
<td>1009</td>
<td>+00:00:03</td>
</tr>
<tr>
<td>povbEEFFA0D56OF.1</td>
<td>LINUX</td>
<td>INTEL</td>
<td>Backfill</td>
<td>Idle</td>
<td>0.200</td>
<td>1009</td>
<td>+00:00:05</td>
</tr>
<tr>
<td>povbEEFFA0DD652.1</td>
<td>LINUX</td>
<td>INTEL</td>
<td>Backfill</td>
<td>Idle</td>
<td>0.200</td>
<td>1009</td>
<td>+00:00:03</td>
</tr>
</tbody>
</table>
```

--More-- (byte 1682)
The **condor_status** command shows only resources available in the local pool. If flocking is configured, it will be possible for jobs to utilize resources in other pools, but those resources are not shown by **condor_status** unless specifically requested. For example, UWM also has access to a large HTCondor pool at the UW Madison campus via flocking.

If a job cluster exceeds the capacity of UWM’s local pool, it will automatically flock processes to the UW Madison pool, rather than hold them in the queue until UWM resources become available.

However, the status of the UW Madison pool will not appear in a simple **condor_status** command. Instead, we must specify the host name of the scheduler host of the remote pool:

```
FreeBSD login.peregrine bacon ~/Hello-world-sh 460: condor_status -pool cm.chtc.wisc.edu
[ about 14,000 lines of output removed ]
slot1_4@wid-003.ch LINUX X86_64 Claimed Busy 1.000 2048 0+00:02:55
slot1_5@wid-003.ch LINUX X86_64 Claimed Busy 1.000 2048 0+00:45:30
slot1_6@wid-003.ch LINUX X86_64 Claimed Busy 1.000 2048 0+00:45:29
slot1_7@wid-laptop.c LINUX X86_64 Claimed Busy 1.000 2048 0+00:45:29
vm-crane.chtc.wisc WINDOWS X86_64 Unclaimed Idle 0.060 1867 19+17:36:22
```

13.1.4 Job Submission

The purpose of this section is to provide the reader a quick start in job scheduling using the most common and useful tools. The full details of job submission are beyond the scope of this document.

### Submit Description Files

Submitting jobs involves specifying a number of job cluster parameters such as the number of cores, the job cluster name (which is displayed by qstat), the name(s) of the output file(s), etc.

In order to record all of this information and make it easy to resubmit the same job cluster, this information is incorporated into a **submit description file**. Using a submit description file saves you a lot of typing when you want to run-submit the same job cluster, and also fully documents the job cluster parameters.

An HTCondor submit description file is a file containing a number of HTCondor variable assignments and commands that indicate the resource requirements of the program to be run, the number of instances to run in parallel, the name of the executable, etc.

**Note**

Unlike the submission scripts of other schedulers such as LSF, PBS, and SLURM, a condor submit description file is not a Unix shell script and hence is not executed on the compute hosts. It is a special type of file specific to HTCondor.

Recall that in most other schedulers, the job cluster parameters are embedded in the script as specially formatted comments beginning with "#SBATCH", "#PBS", or something similar. In HTCondor, the job cluster parameters and the script or program to be run on the compute hosts are kept in separate files.

HTCondor pools are often heterogeneous, i.e. the hosts in the HTCondor pool may run different operating systems, have different amounts of RAM, etc. For this reason, resource requirements are a must in many HTCondor description files.

Furthermore, HTCondor pools typically do not have any shared storage available to all hosts, so the executable, the input files, and the output files are often transferred to and from the execute hosts. Hence, the names of these files must be specified in the submit description file.

Suppose we have the following text in a file called `ls.condor`:
# Sample HTCondor submit description file.

# Use \ to continue an entry on the next line.

# You can query your jobs by command:
# condor_q

# Choose which universe you want your program is running with
# Available options are
#
# - standard:
#   Defaults to transfer executables and files.
#   Use when you are running your own script or program.
#
# - vanilla:
#   Explicitly enable file transfer mechanisms with
#   'transfer_executable', etc.
#   Use when you are using your own files and some installed on the
#   execute hosts.
#
# - parallel:
#   Explicitly enable file transfer mechanism. Used for MPI jobs.

universe = vanilla

# Macros (variables) to use in this submit description file
Points = 1000000000
Process_count = 10

# Specify the executable filename. This can be a binary file or a script.
# NOTE: The POVB execute hosts currently support 32-bit executables only.
# If compiling a program on the execute hosts, this script should compile
# and run the program.
#
# In template.sh, be sure to give the executable a different
# name for each process, since multiple processes could be on the same host.
# E.g. cc -O -o prog.$(Process) prog.c

executable = template.sh

# Command-line arguments for executing template.sh
# arguments =

# Set environment variables for use by the executable on the execute hosts.
# Enclose the entire environment string in quotes.
# A variable assignment is var=value (no space around =).
# Separate variable assignments with whitespace.

environment = "Process=$(Process) Process_count=$(Process_count) Points=$(Points)"

# Where the standard output and standard error from executables go.
# $(Process) is current job ID.

# If running template under both PBS and HTCondor, use same output
# names here as in template-run.pbs so that we can use the same
# script to tally all the outputs from any run.

output = template.out-$\{Process\}
error = template.err-$\{Process\}

# Logs for the job, produced by HTCondor. This contains output from
# HTCondor, not from the executable.
log = template.log

# Custome job requirements
# HTCondor assumes job requirements from the host submitting job.
# IT DOES NOT DEFAULT TO ACCEPTING ANY ARCH OR OPSYS!!
# For example, if the jobs is submitted from peregrine, target.arch is
# "X86_64" and target.opsys is "FREEBSD8", which do not match
# POVB execute hosts.
# You can query if your submitting host is accepted by command:
# condor_q -analyze

# Memory requirements in megabytes
request_memory = 50

# Requirements for a binary compiled on CentOS 4 (POVB hosts):
# requirements = (target.arch == "INTEL") && (target.opsys == "LINUX")

# Requirements for a Unix shell script or Unix program compiled on the
# execute host:
requirements = ((target.arch == "INTEL") || (target.arch == "X86_64")) && 
    ((target.opsys == "FREEBSD") || (target.opsys == "LINUX"))

# Requirements for a job utilizing software installed via FreeBSD ports:
# requirements = ((target.arch == "INTEL") || (target.arch == "X86_64")) && 
# (target.opsys == "FREEBSD")

# Match specific compute host names
# requirements = regexp("compute-s.*.meadows", Machine)

# Explicitly enable executable transfer mechanism for vanilla universe.
# true | false
transfer_executable = true

# yes | no | if_needed
should_transfer_files = if_needed

# All files to be transferred to the execute hosts in addition to the
# executable. If compiling on the execute hosts, list the source file(s)
# here, and put the compile command in the executable script.
transfer_input_files = template.c

# All files to be transferred back from the execute hosts in addition to
# those listed in "output" and "error".
# transfer_output_files = file1,file2,...

# on_exit | on_exit_or_evict
when_to_transfer_output = on_exit

# Explicitly enable executable transfer mechanism for vanilla universe.
# Specify how many jobs you would like to submit to the queue.

```
queue ${Process_count}
```

The description file is submitted to the HTCondor scheduler as a command line argument to the `condor_submit` command:

```
shell-prompt: condor_submit ls.condor
```

Since an HTCondor submit description file specifies a single executable (unlike the submit scripts of some other schedulers such as PBS and LSF), a separate shell script is often used as the executable for the HTCondor job cluster. Below is the file `ls.sh` specified in the HTCondor submit description file.

```
#!/bin/sh
hostname
ls
```

**Note**

Since many HTCondor pools are heterogeneous, you must make sure that the executable is portable, or that you specify resource requirements to ensure that it will only be dispatched to hosts on which it will work.

Using Bourne shell scripts (not bash, ksh, csh, or tcsh) will maximize the portability of a shell script, as discussed in Chapter 8. If you must use different shell, be sure to use a portable shebang line (e.g. `#!/usr/bin/env bash`, not `#!/bin/bash`).

---

**Caution**

Since the software installed on HTCondor pool hosts is highly variable, it's best to use Bourne shell scripts rather than bash, csh, ksh, etc. for jobs submitted to any kind of Unix systems. Recall from Chapter 8 that Bourne shell is the only shell that is sure to be present on every Unix system.

The `condor_submit` command, which is part of the HTCondor scheduler, finds a free core on a execute host, reserves it, transfers the executable and input files from the submit host to the execute host if necessary, and then runs the executable on the execute host.

The executable named in the description is dispatched to the host(s) containing the core(s) allocated by HTCondor, using `ssh`, `rsh` or any other remote shell HTCondor is configured to use.

**Common Features**

The example HTCondor script above shows some of the common variables and other features, which are briefly described below.

- Anything from a `#` character to the end of a line is considered a comment, and is ignored by HTCondor.
- universe: Categorizes jobs in order to set reasonable defaults for different job types and thus reduce the number of explicit settings in the HTCondor script.
- executable: Name of the program or script to run on the execute host(s). This is often transferred from the submit host to the execute hosts, or even compiled on the execute hosts at the start of the job.
- output: Name of a file to which the standard output (normal terminal output) is redirected. Jobs have no connection to the terminal while they run under HTCondor, so the output must be saved to a file for later viewing.
- error: Name of a file to which the standard error (error messages normally sent to the terminal) is redirected.
- input: Name of a file from which to redirect input the program would expect from the standard input (normally the keyboard).
- log: Name of a file to which HTCondor saves its own informative messages about the job cluster.
• requirements: Used to describe the requirements for running the program, such as operating system, memory needs, CPU type, etc.

• transfer_executables = yes/no: Indicate whether the executable specified with the executable variable must be transferred to the execute host before the job executes.

• should_transfer_files = yes/no: Indicate whether input and output files must be transferred to/from the execute host. Normally yes unless a shared file system is available.

• when_to_transfer_output_files: Indicates when output files should be transferred from the execute host back to the submit host. Normally use "on_exit".

• transfer_input_files = list: Comma-separated list of input files to transfer to the execute host.

• queue [count]: Command to submit the job cluster to one or more execute hosts. If no count is given, if defaults to 1 host.

**HTCondor Resource Requirements**

When using a grid, it is important to develop a feel for the resources required by your jobs, and inform the scheduler as accurately as possible what will be needed in terms of CPU time, memory, etc.

You may not know this the first time you run a given job, but after examining the log files from one or more runs, you will have a pretty good idea.

This allows the scheduler to maximize the utilization of precious resources and thereby provide the best possible run times for all users.

HTCondor resource requirements are specified with the requirements variable. There are many parameters available to specify the operating system, memory requirements, etc. Users should try to match requirements as closely as possible to the actual requirements of the job cluster.

For example, if your job requires only 300 megabytes of RAM, then by specifying this, you can encourage HTCondor to save the hosts with several gigabytes of RAM for the jobs that really need them.

**Batch Serial Jobs**

A batch serial submit description file need only specify basic information such as the executable name, input and output files, and basic resource requirements.

```plaintext
universe = vanilla
executable = ls.sh
output = $(Process).stdout
error = $(Process).stderr
log = ls.log
transfer_executable = yes
should_transfer_files = yes
when_to_transfer_output = on_exit
queue 1
```

**Batch Parallel Jobs (Job Clusters)**

A job cluster is a set of independent processes all started by a single job submission.

A batch parallel submit description file looks almost exactly like a batch serial file, but indicates a process count greater than 1 following the queue command:
MPI (Multi-core) Jobs

Most pools are not designed to run MPI jobs. MPI jobs often require a fast dedicated network to accommodate extensive message passing. A pool implemented on lab or office PCs is not suitable for this, a typical local area network in a lab is not fast enough to offer good performance, and MPI traffic may in fact overload it, causing problems for other users.

HTCondor does provide facilities for running MPI jobs, but they are usually only useful where HTCondor is employed as the scheduler for a cluster.

A Submit Description File Template

Below is a sample submit description file with typical options and extensive comments. This script is also available in /share1/Examples on Peregrine.

```plaintext
universe = vanilla
executable = ls.sh
output = $(Process).stdout
error = $(Process).stderr
log = ls.log
transfer_executable = yes
should_transfer_files = yes
when_to_transfer_output = on_exit
queue 10
```

MPI (Multi-core) Jobs

Most pools are not designed to run MPI jobs. MPI jobs often require a fast dedicated network to accommodate extensive message passing. A pool implemented on lab or office PCs is not suitable for this, a typical local area network in a lab is not fast enough to offer good performance, and MPI traffic may in fact overload it, causing problems for other users.

HTCondor does provide facilities for running MPI jobs, but they are usually only useful where HTCondor is employed as the scheduler for a cluster.

A Submit Description File Template

Below is a sample submit description file with typical options and extensive comments. This script is also available in /share1/Examples on Peregrine.

```plaintext
# Sample HTCondor submit description file.
#
# Use \ to continue an entry on the next line.
#
# You can query your jobs by command:
#  condor_q

# Choose which universe you want your program is running with
# Available options are
#
# - standard:
#  Defaults to transfer executables and files.
#  Use when you are running your own script or program.
#
# - vanilla:
# - grid:
#  Explicitly enable file transfer mechanisms with
#  'transfer_executable', etc.
#  Use when you are using your own files and some installed on the
#  execute hosts.
#
# - parallel:
#  Explicitly enable file transfer mechanism. Used for MPI jobs.

universe = vanilla

# Macros (variables) to use in this submit description file
Points = 1000000000
Process_count = 10

# Specify the executable filename. This can be a binary file or a script.
```
# NOTE: The POVB execute hosts currently support 32-bit executables only.
# If compiling a program on the execute hosts, this script should compile
# and run the program.
#
# In template.sh, be sure to give the executable a different
# name for each process, since multiple processes could be on the same host.
# E.g. cc -O -o prog.$(Process) prog.c

executable = template.sh

# Command-line arguments for executing template.sh
arguments =

# Set environment variables for use by the executable on the execute hosts.
# Enclose the entire environment string in quotes.
# A variable assignment is var=value (no space around =).
# Separate variable assignments with whitespace.

environment = "Process=$\{Process\} Process_count=$\{Process_count\} Points=$\{Points\}"

# Where the standard output and standard error from executables go.
# $\{(Process)\} is current job ID.

output = template.out-$\{(Process)\}
error = template.err-$\{(Process)\}

# Logs for the job, produced by HTCondor. This contains output from
# HTCondor, not from the executable.

log = template.log

# Custome job requirements
# HTCondor assumes job requirements from the host submitting job.
# IT DOES NOT DEFAULT TO ACCEPTING ANY ARCH OR OPSYS!!!
# For example, if the jobs is submitted from peregrine, target.arch is
# "X86_64" and target.opsys is "FREEBSD8", which do not match
# POVB execute hosts.
#
# You can query if your submitting host is accepted by command:
# condor_q -analyze

# Memory requirements in megabytes
request_memory = 50

# Requirements for a binary compiled on CentOS 4 (POVB hosts):
# requirements = (target.arch == "INTEL") && (target.opsys == "LINUX")

# Requirements for a Unix shell script or Unix program compiled on the
# execute host:
requirements = ((target.arch == "INTEL") || (target.arch == "X86_64")) &&
  ((target.opsys == "FREEBSD") || (target.opsys == "LINUX"))

# Requirements for a job utilizing software installed via FreeBSD ports:
# requirements = ((target.arch == "INTEL") || (target.arch == "X86_64")) &&

# (target.opsys == "FREEBSD")
# Match specific compute host names
# requirements = regexp("compute-s.*.meadows", Machine)
#
# Explicitly enable executable transfer mechanism for vanilla universe.
#
# true | false
transfer_executable = true

# yes | no | if_needed
should_transfer_files = if_needed

# All files to be transferred to the execute hosts in addition to the
# executable. If compiling on the execute hosts, list the source file(s)
# here, and put the compile command in the executable script.
transfer_input_files = template.c

# All files to be transferred back from the execute hosts in addition to
# those listed in "output" and "error".
# transfer_output_files = file1,file2,...

# on_exit | on_exit_or_evict
when_to_transfer_output = on_exit

# Specify how many jobs you would like to submit to the queue.
queue $(Process_count)

13.1.5 Job Status

While your jobs are running, you can check on their status using `condor_q`.

FreeBSD peregrine bacon ~ 533: condor_q

```
-- Submitter: peregrine.hpc.uwm.edu : <129.89.25.224:37668> : peregrine.hpc.uwm.edu
ID  OWNER  SUBMITTED  RUN_TIME  ST  PRI  SIZE  CMD
 63.0  bacon  6/5  13:45  0+00:00:00  I  0  0.0  ls.sh
 64.0  bacon  6/5  13:46  0+00:00:00  I  0  0.0  ls.sh
```

2 jobs; 2 idle, 0 running, 0 held

The ST column indicates job status (R = running, I = idle, C = completed). Run `man condor_q` for full details.

13.1.6 Terminating a Job or Cluster

If you determine that a job is not behaving properly (by reviewing partial output, for example), you can terminate it using `condor_rm`, which take a job ID or job cluster ID as a command line argument.

To terminate a single job within a cluster, use the job ID form cluster-id,job-index:

FreeBSD peregrine bacon ~ 534: condor_rm 63.0
Job 63.0 marked for removal
FreeBSD peregrine bacon ~ 535: condor_q
To terminate an entire job cluster, just provide the job cluster ID:

FreeBSD peregrine bacon ~ 534: condor_rm 63

### 13.1.7 Job Sequences

If you need to submit a series of jobs in sequence, where one job begins after another has completed,

It’s important to make sure that the current job completed successfully before submitting the next, to avoid wasting resources. It is up to you to determine the best way to verify that a job was successful. Examples might include grepping the log file for some string indicating success, or making the job create a marker file using the `touch` command after a successful run. If the command used in your job returns a Unix-style exit status (0 for success, non-zero on error), then you can simply use the shell’s exit-on-error feature to make your script exit when any command fails. Below is a template for scripts that might run a series of jobs.

```bash
#!/bin/sh
condor_submit job1.condor
condor_wait job1-log-file
# Verify that job1 completed successfully, by the method of your choice
if ! test-to-indicate-job1-succeeded; then
  exit 1
fi
condor_submit job2.condor
```

### 13.1.8 Self-test

1. What command is used to check the status of an HTCondor pool?
2. What kind of file is used to describe an HTCondor job? Is this file a shell script?
4. How can you check the status of your HTCondor jobs?
5. How can you terminate a running HTCondor job?
6. Write and submit a batch-serial HTCondor script called `list-etc.condor` that prints the host name of the execute host on which it runs and a long-listing of `/etc` directory on that host.
   
   The script should store the output of the commands in `list-etc.stdout` and error messages in `list-etc.stderr` in the directory from which the script was submitted.
   
   Quickly check the status of your job after submitting it.

7. Copy your `list-etc.condor` script to `list-etc-parallel.condor`, and modify it so that it runs the `hostname` and `ls` commands on 10 cores instead of just one.
   
   The job should produce a separate output file for each process named `list-etc-parallel.o<jobid>` and a separate error file for each process named `list-etc-parallel.e<jobid>`.
   
   Quickly check the status of your job after submitting it.
13.2 Local Customizations

TBD
Chapter 14

Job Scheduling with PBS (TORQUE)

Before You Begin
Before reading this chapter, you should be familiar with basic Unix concepts (Chapter 7), the Unix shell (Section 7.4.3, redirection (Section 7.17.1), shell scripting (Chapter 8) and job scheduling (Chapter 11.

For complete information, see the official TORQUE documentation at http://www.adaptivecomputing.com/resources/docs/torque/3-0-2/index.php.

14.1 The PBS Scheduler

14.1.1 Cluster Status

The goal of this section is to provide a quick introduction to the PBS scheduler. Specifically, this section covers the current mainstream implementation of PBS, known as TORQUE.

Before scheduling any jobs on through PBS, it is often useful to check the status of the nodes. Knowing how many cores are available may influence your decision on how many cores to request for your next job.

For example, if only 50 cores are available at the moment, and your job requires 200 cores, the job will have to wait in the queue until 200 cores are free. You may end up getting your results sooner if you reduce the number of cores to 50 or less so that the job can begin right away.

The `pbsnodes` command shows information about the nodes in the cluster. This can be used to determine the total number of cores in the cluster, cores in use, etc.

```
FreeBSD peregrine bacon ~/Facil 408: pbsnodes|more
compute-01
  state = job-exclusive
  np = 12
  ntype = cluster
  status = rectime=1331737185, varattr=, jobs=1173[20].peregrine.hpc.uwm.edu, state=free, netload= 15201, gres=, loadave=2.00, ncpus=12, phymem=32515040kb, availmem= 15201, totmem= 15201, idletime=6636140, nusers=3, nsessions= 72397 13348 1005, uname=FreeBSD compute-01.local 8.2-RELEASE-p3 FreeBSD 8.2-RELEASE-p3 #0: Tue Sep 27 18:45:57 UTC 2011 root@amd64-builder.daemonology.net:/usr/obj/usr/src/sys/ GENERIC amd64, csys=freebsd5
```
mom_service_port = 15002
mom_manager_port = 15003
gpus = 0

As a convenience, a script called \texttt{cluster-load} is installed on Peregrine. This script extracts information from \texttt{pbsnodes} and displays the current cluster load in an abbreviated, easy-to-read format.

FreeBSD peregrine bacon ~/Facil 406: cluster-load
Nodes in use: 5
Cores in use: 60
Total cores: 96
Free cores: 36
Load: 62%

The Ganglia Resource Monitor

In addition to the command line tools used to control and monitor jobs, there are also web-based tools for monitoring clusters and grids in a more visual manner.

The \textit{Ganglia Resource Monitor} is a web-based monitoring tool that provides statistics about a collection of computers on a network.

The status of the student cluster, Peregrine, can be seen at \url{http://www.peregrine.hpc.uwm.edu/ganglia/}.

The status of the research cluster, Avi, can be seen at \url{http://www.avi.hpc.uwm.edu/ganglia/}.

14.1.2 Job Status

You can check on their status of running jobs using \texttt{qstat}.

\begin{tabular}{lcccc}
\texttt{peregrine: qstat} & \texttt{Job id} & \texttt{Name} & \texttt{User} & \texttt{Time} & \texttt{Use} & \texttt{S} & \texttt{Queue} \\
\hline
52[].peregrine & \ldotsallel-example bacon & \ldots & \ldots & \ldots & \ldots & \ldots & \ldots \\
\end{tabular}

The Job id column shows the numeric job ID. [ ] indicate a job array. The S column shows the current status of the job. The most common status flags are ‘Q’ for queued (waiting to start), ‘R’ for running, and ‘C’ for completed.

The scheduler retains job status information for a short time after job completion. The amount of time is configured by the systems manager, so it will be different on each site.

The \texttt{qstat} –f flag requests more detailed information. Since it produces a lot of output, it is typically piped through more and/or used with a specific job-id:
The `qstat` command has many flags for controlling what it reports. Run `man qstat` for full details.

### 14.1.3 Using `top`

Using the output from `qstat -f`, we can see which compute nodes are being used by a job.

We can then examine the processes on a given node using a remotely executed `top` command:

```
peregrine: ssh -t compute-003 top
```

**Note** The `-t` flag is important here, since it tells ssh to open a connection with full terminal control, which is needed by `top` to update your terminal screen.

On peregrine, a convenience script is provided to save typing:

```
peregrine: topnode 003
```

### 14.1.4 Job Submission

The purpose of this section is to provide the reader a quick start in job scheduling using the most common and useful tools. The full details of job submission are beyond the scope of this document.

**Submission Scripts**

Submitting jobs involves specifying a number of job parameters such as the number of cores, the job name (which is displayed by `qstat`), the name(s) of the output file(s), etc.
In order to record all of this information and make it easy to resubmit the same job, this information is usually incorporated into a submission script. Using a script saves you a lot of typing when you want to run-submit the same job, and also fully documents the job parameters.

A submission script is an ordinary shell script, with some directives inserted to provide information to the scheduler. For PBS, the directives are specially formatted shell comments beginning with "#PBS".

Suppose we have the following text in a file called hostname.pbs:

```bash
#!/usr/bin/env bash

# A PBS directive
#PBS -N hostname

# A command to be executed on the scheduled node(s)
# Prints the host name of the node running this script.
hostname
```

The script is submitted to the PBS scheduler as a command line argument to the `qsub` command:

```
peregrine: qsub hostname.pbs
```

The `qsub` command, which is part of the PBS scheduler, finds a free core on a compute node, reserves it, and then runs the script on the compute node using `ssh` or some other remote execution command.

Comments beginning with #PBS are interpreted as directives by `qsub` and as any other comment by the shell. Recall that the shell ignores anything on a line after a `#`.

The command(s) in the script are dispatched to the node(s) containing the core(s) allocated by PBS, using `ssh`, `rsh` or any other remote shell PBS is configured to use.

The directives within the script provide command line flags to `qsub`. For instance, the line

```
#PBS -N hostname
```

causes `qsub` to behave as if you had typed

```
peregrine: qsub -N hostname hostname.pbs
```

By putting these comments in the script, you eliminate the need to remember them and retype them every time you run the job. It’s generally best to put all `qsub` flags in the script rather than type any of them on the command line, so that you have an exact record of how the job was started. This will help you determine what went wrong if there are problems, and allow you to reproduce the results at a later time without worrying about whether you did something different.

The script itself can be any valid Unix script, using the shell of your choice. Since all Unix shells interpret the `#` as the beginning of a comment, the #PBS lines will be interpreted only by qsub, and ignored by the shell.

**Note** If you want to disable a #PBS comment, you can just add another `#` rather than delete it. This will allow you to easily enable it again later.

```
#PBS This line is ignored by qsub
#PBS This line is interpreted by qsub
```

It’s a good idea to use a modern shell such as `bash`, `ksh`, or `tcsh`, simply because they are more user-friendly than `sh` or `csh`.

**Practice Break**

Type in the `hostname.pbs` script shown above and submit it to the scheduler using `qsub`. Then check the status with `qstat` and view the output and error files.
Common Flags

```bash
#PBS -N job-name
#PBS -o standard-output-file (default = <job-name>.o<job-id>)
#PBS -e standard-error-file (default = <job-name>.e<job-id>)
#PBS -l resource-requirements
#PBS -M email-address
#PBS -t first-last
```

The `-N` flag gives the job a name which will appear in the output of `qstat`. Choosing a good name makes it easier to keep tabs on your running jobs.

The `-o` and `-e` flags control the name of the files to which the standard output and standard error of the processes are redirected. If omitted, a default name is generated using the job name and job ID.

The `-l` flag is used to specify resource requirements for the job, which are discussed in Section 14.1.4.

The `-t` flag indicates the starting and ending subscripts for a job array. This is discussed in Section 14.1.4.

PBS Resource Requirements

When using a cluster, it is important to develop a feel for the resources required by your jobs, and inform the scheduler as accurately as possible what will be needed in terms of CPU time, memory, etc.

This allows the scheduler to maximize the utilization of precious resources and thereby provide the best possible run times for all users.

If a user does not specify a given resource requirement, the scheduler uses default limits. Default limits are set low, so that users are encouraged to provide an estimate of required resources for all non-trivial jobs. This protects other users from being blocked by long-running jobs that require less memory and other resources than the scheduler would assume.

PBS resource requirements are specified with the `qsub -l` flag.

```bash
#PBS -l procs=count
#PBS -l nodes=node-count:ppn=procs-per-node
#PBS -l cput=seconds
#PBS -l cput=hours:minutes:seconds
#PBS -l vmem=size[kb|mb|gb|tb]
#PBS -l pvmem=size[kb|mb|gb|tb]
```

The `procs` resource indicates the number of processors (cores) used by the job. This resource must be specified for parallel jobs such as MPI jobs, which are discussed in Section 14.1.4. Count cores are allocated by the scheduler according to its own policy configuration. Some cores may be on the same node, depending on the current load distribution.

The `nodes` and `ppn` resources allow the user to better control the distribution of cores allocated to a job. For example, if you want 8 cores all on the same node, you could use `-l nodes=1:ppn=8`. If you want 20 cores, each on a different node, you could use `-l nodes=20:ppn=1`. The best distribution depends on the nature of both your own job and other jobs running at the time. Spreading a job across more nodes will generally reduce memory and disk contention between processes within the job, but also increase communication cost between processes in an MPI job.

Note that specifying a `ppn` value equal to the number of cores in a node ensures exclusive access to that node while the job is running. This is especially useful for jobs with high memory or disk requirements, where we want to avoid contending with other users’ jobs.

The `cput` resource limits the total CPU time used by the job. Most jobs should specify a value somewhat higher than the expected run time simply to prevent program bugs from consuming excessive cluster resources. As noted in the example above, CPU time may be specified either as a number of seconds, or in the format `HH:MM:SS`.

Memory requirements can be specified either as the total memory for the job (`vmem`) or as memory per process within the job (`pvmem`). The `pvmem` flag is generally more convenient, since it does not have to be changed when `procs`, `nodes`, or `ppn` is changed.
Note It is very important to specify memory requirements accurately in all jobs, and it is generally easy to predict based on previous runs by monitoring processes within the job using top or ps. Failure to do so could block other jobs running, even though the resources it requires are actually available.

Batch Serial Jobs

A batch serial submission script need only have optional PBS flags such as job name, output file, etc. and one or more commands.

```bash
#!/usr/bin/env bash
#PBS -N hostname
hostname
```

For simple serial jobs, a convenience script called qsubw is provided. This script submits a job, waits for the job to complete, and then immediately displays the output files. This type of operation is convenient for compiling programs and performing other simple tasks that must be scheduled, but for which we intend to wait for completion and immediately perform the next step.

Caution As currently implemented, qsubw ignores #PBS lines in the submit script containing -N, -o and -e. The entire line is ignored, including other flags. Hence, these flags should not be combined in the same line with other flags in any scripts run through qsubw.

Batch Parallel Jobs (Job Arrays)

A job array is a set of independent processes all started by a single job submission. The entire job array can be treated as a single job by PBS commands such as qstat and qdel, but individual processes are also jobs in and of themselves, and can therefore be manipulated individually via PBS commands.

A batch parallel submission script looks almost exactly like a batch serial script, but requires just one additional flag:

```bash
#!/usr/bin/env bash
#PBS -N hostname-parallel
#PBS -t 1-5
dirname
```

The -t flag is followed a list of integer array IDs. The specification of array IDs can be fairly sophisticated, but is usually a simple range.

The example above requests a job consisting of 5 identical processes which will all be under the job name hostname-parallel. Each process within the job produces a separate output file with the array ID as a suffix. For example, process 2 produces an output file named hostname-parallel.o51-2. (51 is the job ID, and 2 is the array ID within the job.)
Caution The syntax \(-t \ N\) does not work as advertised in the official TORQUE documentation, which states that it is the same as \(-t \ 0-\ N\). In reality, it creates a single job with subscript \(N\).

Practice Break Copy your \(hostname.pbs\) to \(hostname-parallel.pbs\), modify it to run 5 processes, and submit it to the scheduler using \(qsub\). Then check the status with \(qstat\) and view the output and error files.

MPI (Multi-core) Jobs

Scheduling MPI jobs is actually much like scheduling batch serial jobs. This may not seem intuitive at first, but once you understand how MPI works, it makes more sense.

MPI programs cannot be executed directly from the command line as we do with normal programs and scripts. Instead, we must use the \(mpirun\) command to start up MPI programs.

\[
\text{mpirun [mpirun flags] mpi-program [mpi-program arguments]}
\]

Caution Like any other command used on a cluster or grid, \(mpirun\) must not be executed directly from the command line, but instead must be used in a scheduler submission script.

Hence, unlike batch parallel jobs, the scheduler does not directly dispatch all of the processes in an MPI job. Instead, the scheduler dispatches a single \(mpirun\) command, and the MPI system takes care of dispatching and managing all of the MPI processes that comprise the job.

Since the scheduler is only dispatching one process, but the MPI job may dispatch others, we must add one more item to the submit script to inform the scheduler how many processes MPI will dispatch. In PBS, this is done using a resource specification, which consists of a \(-l\) followed by one or more resource names and values.

```
#!/bin/sh
#PBS -N MPI-Example
#PBS -l procs=48
mpirun mpi-program
```

When running MPI jobs, it is often desirable to have as many processes as possible running on the same node. Message passing is generally faster between processes on the same node than between processes on different nodes, because messages passed within the same node need not cross the network. If you have a very fast network such as Infiniband or 10 gigabit Ethernet, the difference may be marginal, but on more ordinary networks such as gigabit Ethernet, the difference can be enormous.

Environment Variables

PBS sets a number of environment variables when a job is started. These variables can be used in the submission script and within other scripts or programs executed as part of the job.

One of the most important is the \(PBS_O_WORKDIR\) variable. By default, PBS runs processes on the compute nodes with the home directory as the current working directory. Most jobs, however, are run from a project directory under or outside the home
directory that is shared by all the nodes in the cluster. Usually, the processes on the compute nodes should all run in the same project directory. In order to ensure this, we can either use the –d flag, or add this command to the script before the other command(s):

```
cd $PBS_O_WORKDIR
```

$PBS_O_WORKDIR is set by the scheduler to the directory from which the job is submitted. Note that this is the directory must be shared by the submit node and all compute nodes via the same pathname: The scheduler takes the pathname from the submit node and then attempts to cd to it on the compute node(s).

Using this variable is more convenient than using the –d flag, since we would have to specify a hard-coded path following –d. If we move the project directory, scripts using –d would have to be modified, while those using `cd $PBS_O_WORKDIR` will work from any starting directory.

The PBS_JOBNAME variable can be useful for generating output filenames within a program, among other things.

The PBS_ARRAYID variable is especially useful in jobs arrays, where each job in the array must use a different input file and/or generate a different output file. This scenario is especially common in Monte Carlo experiments and parameter sweeps, where many instances of the same program are run using a variety of inputs.

```
#!/usr/bin/env bash
#PBS -t 1-100

cd $PBS_O_WORKDIR
./myprog input-$PBS_ARRAYID.txt
```

A Submit Script Template

Below is a sample submit script with typical options and extensive comments. This template is also available as a macro in APE (Another Programmer’s Editor) and in /share1/Examples on Peregrine.

```
#!/bin/sh

#########################################################################
# Torque (PBS) job submission script template
# "#PBS" denotes PBS command-line flags
# "##PBS" is ignored by torque/PBS
#########################################################################

# Job name that will be displayed by qstat, used in output filenames, etc.
#PBS -N pbs-template

# Job arrays run the same program independently on multiple cores.
# Each process is treated as a separate job by PBS.
# Each job has the name pbs-template[index], where index is
# one of the integer values following -t. The entire array can
# also be treated as a single job with the name "pbs-template[]".
##PBS -t 1-10 # 10 jobs with consecutive indexes
##PBS -t 2,4,5,6 # Explicitly list arbitrary indexes

# Specifying cores and distribution across nodes
# Arbitrary cores: the most flexible method. Use this for all jobs with
# no extraordinary requirements (e.g. high memory/process). It gives the
# scheduler the maximum flexibility in dispatching the job, which could
# allow it to start sooner.

```bash
##PBS -l procs=6
```

# Specific number of cores/node. Use this for high-memory processes, or any other time there is a reason to distribute processes across multiple nodes.

# For multicore jobs (e.g. MPI), this requirement is applied once to the entire job. To spread an MPI job out so that there is only one process per node, use:

```bash
##PBS -l nodes=8:ppn=1
```

# For job arrays, it is applied to each job in the array individually.

E.g.

```bash
##PBS -l nodes=1:ppn=N
```

where N is the total number of cores on a node. This reserves an entire node for each job in the array, i.e. other jobs will not be able to use any cores on that node. Useful for high-memory jobs that need all the memory on each node.

```bash
##PBS -l nodes=3:ppn=1
```

# Specifying virtual memory requirements for each process within the job.

This should be done for all jobs in order to maximize utilization of cluster resources. The scheduler will assume a small memory limit unless told otherwise.

```bash
##PBS -l pvmem=250mb
```

# CPU time and wall time. These should be specified for all jobs.

Estimate how long your job should take, and specify 1.5 to 2 times as much CPU and wall time as a limit. This is only to prevent programs with bugs from occupying resources longer than they should. The scheduler will assume a small memory limit unless told otherwise.

```bash
##PBS -l cput=seconds or [HH:]MM:SS
##PBS -l walltime=seconds or [HH:]MM:SS
```

# Environment variables set by the scheduler. Use these in the commands below to set parameters for array jobs, control the names of output files, etc.

```bash
##PBS_O_HOST the name of the host where the qsub command was run
##PBS_JOBID the job identifier assigned to the job by the batch system
##PBS_JOBNAME the job name supplied by the user.
##PBS_WORKDIR the absolute path of the current working directory of the qsub command
##PBS_NODEFILE name of file containing compute nodes
```
### Shell commands

# Torque starts from the home directory on each node by default, so we
# must manually cd to the working directory to ensure that output files
# end up in the project directory, etc.

```
cd $PBS_O_WORKDIR
```

# Optional preparation example:
# Remove old files, alter PATH, etc. before starting the main process

```
rm output*.txt
PATH=/usr/local/mpi/openmpi/bin:${PATH}
export PATH
```

# MPI job example:
# mpirun ./pbs-template

# Serial or array job example:
#. ./pbs-template -o output-$PBS_JOBID.txt

### 14.1.5 Terminating a Job

If you determine that a job is not behaving properly (by reviewing partial output, for example), you can terminate it using `qdel`, which takes a job ID as a command line argument.

```
peregrine: qstat
<table>
<thead>
<tr>
<th>Job id</th>
<th>Name</th>
<th>User</th>
<th>Time</th>
<th>Use</th>
<th>S</th>
<th>Queue</th>
</tr>
</thead>
<tbody>
<tr>
<td>53.peregrine</td>
<td>MPI-Benchmark</td>
<td>bacon</td>
<td>0</td>
<td>R</td>
<td></td>
<td>batch</td>
</tr>
</tbody>
</table>
peregrine: qdel 53
```

```
peregrine: qstat
<table>
<thead>
<tr>
<th>Job id</th>
<th>Name</th>
<th>User</th>
<th>Time</th>
<th>Use</th>
<th>S</th>
<th>Queue</th>
</tr>
</thead>
<tbody>
<tr>
<td>53.peregrine</td>
<td>MPI-Benchmark</td>
<td>bacon</td>
<td>00:04:20</td>
<td>C</td>
<td></td>
<td>batch</td>
</tr>
</tbody>
</table>
```

### 14.1.6 Viewing Output of Active Jobs

As mentioned in Section 14.1.4, the output sent by processes to standard output and standard error is redirected to files, as named

```
However, these files are not created until the job ends. Output is stored in temporary files until then.
A convenience script, called `qpeek`, is provided for viewing the output of a job stored in the temporary files. The `qpeek` is not part of PBS, but is provided as an add-on. It takes a single job ID as an argument.

```
FreeBSD peregrine bacon ~ 499: qsub solveit.pbs
297.peregrine.hpc.uwm.edu
FreeBSD peregrine bacon ~ 500: qpeek 297
Computing the solution...
1 2 3 4 5
FreeBSD peregrine bacon ~ 501:
```
14.1.7 Self-test

1. What is the PBS command for showing the current state of all nodes in a cluster?

2. What is the PBS command to show the currently running jobs on a cluster?

3. Write and submit a batch-serial PBS script called list-etc.pbs that prints the host name of the compute node on which it runs and a long-listing of /etc directory on that node.
   The script should store the output of the commands in list-etc.stdout and error messages in list-etc.stderr in the directory from which the script was submitted.
   The job should appear in qstat listings under the name "list-etc".
   Quickly check the status of your job after submitting it.

4. Copy your list-etc.pbs script to list-etc-parallel.pbs, and modify it so that it runs the hostname and ls commands on 10 cores instead of just one.
   The job should produce a separate output file for each process named list-etc-parallel.o<jobid>-<arrayid> and a separate error file for each process named list-etc-parallel.e<jobid>-<arrayid>.
   Quickly check the status of your job after submitting it.

5. What is the PBS command for terminating a job with job-id 3545?

6. What is the PBS command for viewing the terminal output of a job with job-id 3545 while it is still running?

7. What is the PBS command for showing detailed job information about the job with job-id 3254?
Chapter 15

Job Scheduling with LSF

Before You Begin
Before reading this chapter, you should be familiar with basic Unix concepts (Chapter 7), the Unix shell (Section 7.4.3, redirection (Section 7.17.1), and shell scripting (Chapter 8).

15.1 The LSF Scheduler

LSF, which stands for Load Sharing Facility, is part of a commercial cluster management suite called Platform Cluster Management. The LSF scheduler allows you to schedule jobs for immediate queuing or to run at a later time. It also allows you to monitor the progress of your jobs and manipulate them after they have started. This document describes the basics of using LSF, and how to get more detailed information from the official LSF documentation.

15.1.1 Submitting Jobs

Jobs are submitted to the LSF scheduler using the `bsub` command. The `bsub` command finds available cores within the cluster and executes your job(s) on the selected cores. If the resources required by your job are not immediately available, the scheduler will hold your job in a queue until they become available.

You can provide the Unix command you want to schedule as part of the `bsub` command, but the preferred method of using `bsub` involves creating a simple shell script. Several examples are given below. Writing shells scripts is covered in Chapter 8.

Using `bsub` without a script

```bash
[user@hd1 ~]$ bsub -J hostname_example -o hostname_output%J.txt hostname
```

The `bsub` command above finds an available core and dispatches the Unix command `hostname` to that core. The output of the command (the host name of the node containing the core) is appended to `hostname_output%J.txt`, where `%J` is replaced by the job number. The `-o` flag in the command tells `bsub` that any output the `hostname` command tries to send to the terminal should be appended to the filename following `-o` instead.

The job is given the name `hostname_example` via the `-J` flag. This job name can be used by other LSF commands to refer to the job while it is running. A few of these commands are described below.

Using a Submission Script

The job in the example above can also be submitted using a shell script.
When used with the scheduler, scripts document exactly how the `bsub` command is invoked, as well as the exact sequence of Unix commands to be executed on the cluster. This allows you to re-execute the job in exactly the same manner without having to remember the details of the command. It also allows you to perform preparation steps within the script before the primary command, like removing old files, going to a specific directory, etc.

To script the example above, enter the appropriate Unix commands into a text file, e.g. `hostname.bsub`, using your favorite text editor.

```bash
[users@hd1 ]$ nano hostname.bsub
```

Once you’re in the editor, enter the following text, and then save and exit:

**Example 15.1 Simple LSF Batch Script**

```bash
#!/usr/bin/env bash
#BSUB -J hostname_example -o hostname_output%J.txt
#BSUB -v 1000000
hostname
```

Then submit the job by running:

```bash
[users@hd1 ]$ bsub < hostname.bsub
```

Note that the script is fed to `bsub` using input redirection.

The first line of the script:

```bash
#!/usr/bin/env bash
```

indicates that this script should be executed by bash (Bourne Again shell).

The second and third lines:

```bash
#BSUB -J hostname_example -o hostname_output%J.txt
#BSUB -v 1000000
```

specify command line flags used to be used with the `bsub` command. Note that they’re the same flags we entered on the Unix command line when running `bsub` without a script. You can place any number of command line flags on each `#BSUB` line. It makes no difference to `bsub`, so it’s strictly a matter of readability and personal taste.

Recall that to the shell, anything following a `#` is a comment. Lines that begin with `#BSUB` are specially formatted shell comments that are recognized by `bsub` but ignored by the shell running the script.

The beginning of the line must be exactly `#BSUB` in order to be recognized by `bsub`. Hence, we can disable an option without removing it from the script by simply adding another `#`:

```bash
##BSUB -o hostname
```

The new command-line flag here, `-v 1000000`, limits the memory use of the job to 1,000,000 kilobytes, or 1 gigabyte.

---

**Caution** All jobs should use memory-limits like this to prevent them from accidentally overloading the cluster. Programs that use too much memory can cause compute nodes to crash, which may kill other users’ jobs as well as your own. It’s possible to cause another user to lose several weeks worth of work by simply miscalculating how much memory your job requires.

All lines in the script that are not comments are interpreted by bash as Unix commands, and are executed on the core(s) selected by the scheduler.
15.1.2 Job Types

Scheduled jobs fall under one of several classifications. The differences between these classifications are mainly conceptual and the differences in the commands for submitting them via `bsub` can be subtle.

**Batch Serial Jobs**

The example above is what we call a batch serial job. Batch serial jobs run on a single core, and any output they send to the standard output (normally the terminal) is redirected to the file named using `-o filename` with `bsub`. Batch jobs do not display output to the terminal window, and cannot receive input from the keyboard.

**Interactive Jobs**

If you need to see the output of your job on the screen during execution, or provide input from the keyboard, then you need an interactive job.

From the user’s point of view, an interactive job runs as if you had simple entered the command at the Unix prompt instead of running it under the scheduler. The important distinction is that with a scheduled interactive job, the scheduler decides which core to run it on. This prevents multiple users from running interactive jobs on the same node and potentially swamping the resources of that node.

From the scheduler’s point of view, an interactive job is almost the same as a batch serial, except that the output is not redirected to a file, and interactive jobs can receive input from the keyboard.

To run an interactive job in `bsub`, simply add `-I` (capital i), `-Ip`, or `-Is`. The `-I` flag alone allows the job to send output back from the compute node to your screen. It does not, however, allow certain interactive features required by editors and other full-screen programs. The `-Ip` flag creates a pseudo-terminal, which enables terminal features required for most programs. The `-Is` flag enables shell mode support in addition to creating a pseudo-terminal. `-Ip` should be sufficient for most purposes.

**Example 15.2 Batch Interactive Script**

```bash
#!/usr/bin/env bash

#BSUB -J hostname_example -Ip
hostname
```

Note that the `-o` flag cannot be used with interactive jobs, since output is sent to the terminal screen, not to a file.

**Waiting for Jobs to Complete**

The `bsub` `-K` flag causes `bsub` to wait until the job completes.

This provides a very simple mechanism to run a series of jobs, where one job requires the output of another.

It can also be used in place of interactive jobs for tasks such as program compilation, if you want to save the output for future reference.

**Batch Parallel Jobs**

Batch serial and interactive jobs don’t really make good use of a cluster, since they only run your job a single core. In order to utilize the real power of a cluster, we need to run multiple processes in parallel (at the same time).

If your computations can be decomposed into N completely independent tasks, then you may be able to use a batch parallel job to reduce the run time by nearly a factor of N. This is the simplest form of parallelism, and it maximizes the performance gain on a cluster. This type of parallel computing is often referred to as *embarrassingly parallel*, *loosely coupled*, *high throughput*.
computing (HTC), or grid computing. It is considered distinct from tightly coupled or high performance computing (HPC), where the processes that make up a parallel job communicate and cooperate with each other to complete a task.

In LSF, batch parallel jobs are distinguished from batch serial jobs in a rather subtle way; by simply appending a job index specification to the job name:

**Example 15.3 Batch Parallel Script**

```bash
#!/usr/bin/env bash
#BSUB -J parallel[1-10] -o parallel_output%J.txt
printf "Job $LSB_JOBINDEX running on 'hostname'
" > output_$LSB_JOBINDEX.txt
```

This script instructs the scheduler to allocate 10 cores in the cluster, and start a job consisting of 10 simultaneous processes with names parallel[1], parallel[2], ... parallel[10] within the LSF scheduler. The scheduler allocates 10 cores, and the commands in the script are executed on all 10 cores at (approximately) the same time.

The environment variable $LSB_JOBINDEX is created by the scheduler, and assigned a different value for each process within the job. Specifically, it will be a number between 1 and 10, since these are the subscripts specified with the `-J` flag. This allows the script, as well as commands executed by the script, to distinguish themselves from other processes. This can be useful when you want all the processes to read different input files or store output in separate output files.

Note that the output of the printf command above is redirected to a different file for each job index. Normally, printf displays its output in the terminal window, but this example uses a shell feature called output redirection to send it to a file instead. When the shell sees a `>` in the command, it takes the string after the `>` as a filename, and causes the command to send its output to that file instead of the terminal screen.

After all the processes complete, you will have a series out output files called output_1.txt, output_2.txt, and so on. Although each process in the job runs on a different core, all of the cores have direct access to the same files and directories, so all of these files will be in the same place when the job finishes.

Each file will contain the host name on which the process ran. Note that since each node in the cluster has multiple cores, some of the files may contain the same host name. Remember that the scheduler allocates cores, not nodes.

**MPI Jobs**

As mentioned earlier, if you can decompose your computations into a set of completely independent parallel processes that have no need to talk to each other, then a batch parallel job is probably the best solution.

When the processes within a parallel job must communicate extensively, special programming is required to make the processes exchange information.

MPI (Message Passing Interface) is the de facto standard library and API (Application Programming Interface) for such tightly-coupled distributed programming. MPI can be used with general-purpose languages such as C, Fortran, and C++ to implement complex parallel programs requiring extensive communication between processes. Parallel programming can be very complex, but MPI will make the program implementation as easy as it can be.

There are multiple implementations of the MPI standard, several of which are installed on the cluster. The OpenMPI implementation is the newest, most complete, and is becoming the standard implementation. However, some specific applications still depend on other MPI implementations, so many clusters have multiple implementation installed. If you are building and running your own MPI applications, you may need to select a default implementation using the mpi-selector-menu command. A default selection of openmpi_gcc_qlc is usually configured for new users, but you can change it at any time, and as often as you like.

All MPI jobs are started by using the `mpirun` command, which is part of the MPI installation. The `mpirun` command dispatches the mpi program to all the nodes that you specify with the command, and sets up the communication interface that allows them to pass messages to each other while running.

---

**Caution** On a cluster shared by many users, it is impractical to use `mpirun` directly, since it requires the user to specify which cores to use, and cores should be selected by the scheduler instead. All MPI jobs should use the wrappers described below instead of using `mpirun` directly.
To facilitate the use of `mpirun` under the LSF scheduler, LSF provides wrapper commands for each MPI implementation. For example, to use OpenMPI, the command is `openmpi_wrapper`. These wrappers allow the scheduler to pass information such as the list of allocated cores down to `mpirun`.

**Example 15.4 LSF and OpenMPI**

```bash
#!/usr/bin/env bash

#BSUB -o matmult_output%J.txt -n 10
genopenmpi_wrapper matmult
```

In this script, `matmult` is an MPI program, i.e. a program using the MPI library functions to pass messages between multiple cooperating matmult processes.

The scheduler executes `openmpi_wrapper` on a single core, and `openmpi_wrapper` in turn executes `matmult` on 10 cores.

Note that MPI jobs in LSF are essentially batch serial jobs where the command is `openmpi_wrapper` or one of the other wrappers. The scheduler only allocates multiple cores, and executes the provided command on one of them. From there, MPI takes over.

Although `openmpi_wrapper` executes on only one core, the MPI program as a whole requires multiple cores, and these cores must be allocated by the scheduler. The `bsub` flag `-n 10` instructs the LSF scheduler to allocate 10 cores for this job, even though the scheduler will only dispatch `openmpi_wrapper` to one of them. The `openmpi_wrapper` command then dispatches `matmult` to all of the cores provided by the scheduler, including the one on which `openmpi_wrapper` is running.

Note that you do not need a cluster to develop and run MPI applications. You can develop and test MPI programs on your PC provided that you have a compiler and an MPI package installed. The speedup provided by MPI will be limited by the number of cores your PC has, but the program should work basically the same way on your PC as it does on a cluster, except that you will probably use `mpirun` directly on your PC rather than submit the job to a scheduler.

### 15.1.3 Job Control and Monitoring

#### Listing jobs

After submitting jobs with `bsub`, you can monitor their status with the `bjobs` command.

```bash
[user@hd1 ~]$ bjobs
```

By default, `bjobs` limits the amount of information displayed to fit an 80-column terminal. To ensure complete information is printed for each job, add the `-w` (wide output) flag.

```bash
[user@hd1 ~]$ bjobs -w
```

By default, the `bjobs` command will list jobs you have running, waiting to run, or suspended. It displays basic information such as the numeric job ID, job name, nodes in use, and so on.

If you would like to see what other users are running on the cluster, run

```bash
[user@hd1 ~]$ bjobs -w -u all
```

There are many options for listing a subset of your jobs, other users’ jobs, etc. Run `man bjobs` for a quick reference.

#### Checking Progress

When you run a batch job, the output file may not be available until the job finishes. You can view the output generated so far by an unfinished job using the `bpeek` command:

```bash
[user@hd1 ~]$ bpeek numeric-job-id
```

or

```bash
[user@hd1 ~]$ bpeek -J job-name
```

The job-name above is the same name you specified with the `-J` flag in `bsub`. The numeric job id can be found by running `bjobs`.
Using top

Using the output from `bjobs`, we can see which compute nodes are being used by a job. We can then examine the processes on a given node using a remotely executed `top` command:

```
avi: ssh -t compute-1-03 top
```

*Note* The `-t` flag is important here, since it tells ssh to open a connection with full terminal control, which is needed by `top` to update your terminal screen.

Terminating Jobs

A job can be terminated using the `bkill` command:

```
[user@hd1 ~]$ bkill numeric-job-id
```

or

```
[user@hd1 ~]$ bkill -J job-name
```

Again, the numeric job id can be found by running `bjobs`, and the job-name is what you specified with the `-J` option in `bsub`.

Email Notification

LSF has the ability to notify you by email when a job begins or ends. All notification emails are sent to your PantherLINK account.

To receive an email notification when your job ends, add the following to your `bsub` script:

```
#BSUB -N
```

To receive an email notification when your job begins, add the following to your `bsub` script:

```
#BSUB -B
```

The `-B` flag is only useful when the load in the cluster is too high to accommodate your job at the time it is submitted.

15.2 Good Neighbor Scheduling Strategies

15.2.1 Job Distribution

Modern clusters consist of nodes with many cores. This raises the question of how processes should be distributed across nodes. If independent processes such as serial jobs and job arrays are spread out across as many nodes as possible, there may be less resource contention within each node, leading to better overall performance. Spreading jobs out this way can also be a necessity. For example, if each process in a job array requires 20 gigabytes of RAM, and each node has 24 gigabytes available, then we can’t run more than one of these processes per node.

On the other hand, shared memory jobs require their processes to be on the same node, and multicore jobs (such as MPI jobs) that use a lot of interprocess communication will perform better when their processes are on the same node. (Communication between any two processes is generally faster if they are on the same node.)

When large numbers of serial and job array processes are spread out across the cluster, shared memory jobs may be prevented from starting, because there are no individual nodes with sufficient free cores available, even though there are plenty of free
cores across the cluster. Multicore jobs may see degraded performance in this situation, since their processes are forced to run on different nodes and therefore must use the network to communicate.

Default scheduler policies tend to favor spreading out serial and array jobs across many nodes, and clumping multicore jobs on the same node.

Users running serial and array jobs can take an extra step toward being a good citizen by overriding the default scheduler policies. If you know that your serial or array jobs won’t have a problem with resource contention, you can tell the scheduler to dispatch as many of them as possible to heavily loaded nodes using a simple `bsub` resource specifier:

```bash
#BSUB -R "order([-ut])"
```

This will leave more nodes with a large number of cores available, which will help users running shared memory and multicore jobs.

### 15.3 More Information

For details of any of the LSF commands or any other Unix, run `man command` on the cluster. For example:

```bash
[user@hd1 ~]$ man bsub
```

This will display the man page for the `bsub` command. Most other Unix commands also have man pages. Note that the man pages are meant to provide a quick reference to users who already have a basic understanding of the command. Beginners looking for a more complete explanation of a topic may be better off finding a good book or tutorial.

The LSF User’s Guide is a more complete reference for the LSF scheduler.

The HPC Community website is a gathering place for LSF users. It offers many documents for download as well as a user forum.

Links to the LSF User’s Guide, the HPC Community website, and more can be found at the cluster user documentation page: [http://www4.uwm.edu/hpc/support_resources](http://www4.uwm.edu/hpc/support_resources).
Part III

Programming for Performance
Chapter 16

Software Development

16.1 Programming is NOT Inherently Painful

Most people make most things far more complicated than they need to be. Nowhere will you see this fact documented better than in a typical computer program. Most computer programs could be reduced to a fraction of their present size, made to run orders of magnitude faster, and made far easier to read and maintain. The majority of complexity in most code is due to bad decisions on the part of the programmer.

What this means for you as you begin to learn programming is that the experience does not need to be painful. You may have heard from other students that their computer programming class was really difficult (often paraphrased in more colorful language). If so, their struggle was mostly a result of following bad practices, due to poor teaching or failure to follow their teacher’s advice.

Pain is not inherent in the programming process. If you are taught properly and have enough basic self-discipline to follow those teachings, programming can be relatively easy and fun. Programming is a creative, challenging, and often time-consuming process, but only a fraction as difficult as most people make it seem.

Take your time learning to program and PRACTICE the basics until you understand them well. Master variables before learning conditionals. Master conditionals before learning loops. Master loops before learning subprograms. Master subprograms before learning arrays. And so on... A little time invested early in the learning process will save you a lot of wasted time and frustration later.

The goal of this text is to focus on good practices and lead you down the right path so that your experience as a programming requires minimal effort and produces good results: Simple, fast, readable, easily maintainable program code.

We’ll begin with some important background knowledge and then learn to program one step at a time, exploring relevant best practices and pitfalls along the way.

16.2 Operating System Selection

As discussed in Chapter 7, most research computing is done on Unix-compatible operating systems. Every operating system you are likely to use, with the exception of Microsoft Windows, is Unix-compatible. There are many commercial Unix systems used in corporate data centers, but researchers are most likely to be running Mac OS X or a free Unix system such as one of the many BSD or Linux distributions.

Windows users can run Unix software using a compatibility layer such as Cygwin (Section 7.5.1), or by running a Unix system on a virtual machine (Section 7.22) such as VirtualBox. Preconfigured virtual machine installations are available for many Unix systems, so obtaining a Unix environment on your Windows PC is not difficult.
Cygwin makes it extremely easy to do most Unix code development on a Windows computer. Through Cygwin, you can easily install compilers and interpreters for most common languages, as well as other Unix development tools such as editors, debuggers, etc.

---

**Caution**

Code developed in proprietary Windows development environments such as Visual Studio may be very difficult to port to Unix systems. This is often a major problem for researchers who discover down the road that their PC is not fast enough and want to run their code on high-power Unix servers or clusters. The only option for them is to heavily modify or rewrite the code so that it is Unix-compatible.

Although Apple's Mac OS X is Unix-compatible, the Xcode development environment is proprietary, and projects developed in Xcode are not portable to other Unix systems. If you develop using an Xcode project, you will need to maintain a separate build system for other Unix platforms. Alternatively, you can develop under Mac OS X using a single, portable build systems such as a simple Makefile that will work in all Unix environments (including Cygwin).

---

For most researchers, it makes no difference which operating system they use to develop and test their code. Unix systems are highly compatible with each other and most of the features of any one of them are available in the others.

Each system does have some of its own special features, but most of them are not relevant to most scientific software, so it is generally easy to write code that is portable among all Unix systems.

A very small percentage of researchers may want to use proprietary systems such as CUDA, which run only on a few select Unix systems.

It’s best to think about your basic research needs before you choose an operating system and then decide whether using a non-portable system offers enough benefit to warrant limiting the portability of your code.

There are often more portable alternatives available or under development. For example, OpenCL is a more portable and flexible alternative to CUDA that aims to work on any operating system and any hardware platform. At the time of this writing, however, CUDA still offers better performance. If that added performance might mean meeting a grant deadline that you would miss otherwise, then using CUDA would be a wise decision.

### 16.3 Language Types

Programming languages are categorized into several types, which are described in the following sections.

#### 16.3.1 Machine Language

A machine language is a set of binary codes which is used to direct the activities of a Central Processing Unit (CPU).

Everything a computer does is the result of running machine language instructions. No matter what language you use to program, the computer hardware is ultimately running machine language in order to execute your program.

Individual operations in a machine language are known as *instructions*. Hence, a machine language is also known as an *instruction set*. Each instruction consists of an *operation code (opcode for short)* and possibly some operands. For example, and *add* instruction contains a binary opcode that causes the CPU to initiate a sequence of operations to add two numbers, and usually two or three operands that specify where to get the terms to be added, and where to store the result.

The CPU reads instructions from memory, and the bits in the instruction trigger “switches” in the CPU, causing it to *execute* the instruction. For example, the following is an example of a machine code add instruction for the MIPS microprocessor:

```
0000000010000110001100000100000
```

This instruction would cause the processor to add the contents of registers 2 and 3, and store the result back into register 3. The meaning of each bit in this instruction is depicted in Table 16.1.

A CPU *architecture* is defined by its instruction set. For example, the Intel x86 family of architectures has a specific set of instructions, with variations that have evolved over time (8086, 8088, 80286, 80386, 80486, Pentium, Xeon, Core Duo, AMD...
Table 16.1: Example MIPS Instruction

<table>
<thead>
<tr>
<th>opcode</th>
<th>source1</th>
<th>source2</th>
<th>destination</th>
<th>unused</th>
<th>opcode continued</th>
</tr>
</thead>
<tbody>
<tr>
<td>000000</td>
<td>00010</td>
<td>00011</td>
<td>00011</td>
<td>00000</td>
<td>100000</td>
</tr>
<tr>
<td>add</td>
<td>register 2</td>
<td>register 3</td>
<td>register 3</td>
<td>-</td>
<td>add</td>
</tr>
</tbody>
</table>

Athlon, etc.) The x86 architectures have a completely different instruction set than the MIPS architecture, the ARM architecture, the PowerPC architecture, and so on.

The architecture is separate from the implementation. For example, Intel and AMD (Advanced Micro Devices) are two separate companies that make implementations of the x86 architectures. Some popular Intel implementations include the Core Duo, Xeon, Nehalem, etc. Some AMD x86 implementations include the Athlon, Duron, Sempron, etc. While the circuit diagrams of all these processors are very different from each other, they all implement the same basic x86 instruction set plus some extensions.

The fact that machine language is specific to one architecture presents an obvious problem: Machine language programs written for one architecture have to be completely rewritten in order to run on a different architecture, i.e. they are not portable. Program development and maintenance is enormously time consuming, and therefore expensive. The solution to this problem is discussed in Section 16.3.3.

In addition to the lack of portability, machine language programs tend to be very long, since the machine instructions are quite primitive. Most machine instructions can only perform a single, simple operation such as adding two numbers. It may take a sequence of dozens of instructions to evaluate a simple polynomial.

### 16.3.2 Assembly Language

In order to program in machine language, one would have to memorize or look up binary codes for opcodes and operands in order to read or write the instructions. This process is far too tedious and error prone to allow for productive (or enjoyable) programming.

One of the first things early programmers did to make the job easier is create a mnemonic, or symbolic form of machine language that is easier to read. This new form was named assembly language. For example, instead of writing

```
00000000010000110001100000100000
```

a programmer could write

```
add $3, $2, $3
```

which is obviously much more intuitive.

Assembly language also makes it possible for the programmer to use named variables instead of numeric memory addresses and many other convenient features.

However, the CPU can’t understand this mnemonic form, so it has to be translated, or assembled into machine language before the computer can run it. Hence, it was given the name assembly language.

The set of assembly language instructions is generally a close, but not exact match to the machine language. When designing assembly languages, programmers often add some simple features to make programming a little easier than a 1-to-1 mapping to machine language would provide. For example, some machine instructions may be given more than one assembly instruction name to help make programs more readable, and some assembly instructions may actually translate to a sequence of machine instructions instead of just one. Assembly instructions that don’t translate exactly to one machine instruction are known as pseudo-instructions.

While assembly language is much easier to read and write than machine language, it still suffers from two major problems:

- It is still specific to one architecture, i.e. it is not portable.
- The instructions are still very primitive, so the programs are long and difficult to follow.
16.3.3 High Level Languages (HLLs)

Overview

Early programmers quickly realized that it should be feasible to automate the process of writing certain types of machine or assembly code. Programs often contain sequences of instructions that evaluate mathematical expressions, perform repetitive execution, make decisions, print out numbers and text, etc. Coding such sequences is often a simple, but tedious process.

Wouldn’t it be nice if we had a program that could take a mathematical expression and write the machine code to evaluate it for us? What if it could also write the necessary code to input and output numbers?

In the 1950’s, a team at IBM led by John Backus set out to do just that, and their efforts produced the first widely used high-level language, FORTRAN, which is short for FORmula TRANslator. The program that performed the translation from algebraic expressions and other convenient constructs to machine language was named a compiler.

FORTRAN made it much easier to write programs, since we could now write a one-line algebraic expressions and let the compiler convert it to the long sequence of machine instructions. We could write a single print statement instead of the hundreds of machine instructions needed to perform common tasks like convert a number stored in binary such as 0000000010110100 to the sequence of characters “180” that are sent to our terminal screen.

In addition to making our programs much shorter and easier to understand, FORTRAN paved the way for another major benefit: portability. We could now write programs in FORTRAN, and by modifying the compiler to output machine code for different CPU architectures, we could run the same program on any type of computer.

16.3.4 Self-test

1. Define the following terms in the context of CPU hardware:
   - Machine language
   - Instruction set
   - CPU Architecture
   - Implementation

2. Describe the basic structure of a machine instruction.

3. Discuss three major limitations of machine language.

4. Describe four advantages of programming in assembly language over programming in machine language.

5. What do assembly language and machine language have in common?

6. Describe two advantages of programming in a high-level language over programming in assembly language.

16.4 Language Selection

16.4.1 Decision Factors

Selecting a language can be a difficult and controversial decision. Ultimately, the decision should come down to costs, which may include some or all of the following:

- Hardware
- Software purchases
- Labor (development, testing, maintenance)
- Training
 Operational costs (infrastructure, electricity, etc.)

Which of the above factors dominate the total cost will depend on the application, so language choices will be vary greatly for different situations. Is the main goal to minimize programmer time? To maximize reliability? To maximize execution speed?

For interactive programs, maximizing performance is generally not a major concern, so little effort goes into optimization. Users don’t usually care whether a program responds to their request in 1/2 second or 50 times faster - 1/100 second.

In High Performance Computing, on the other hand, the primary goal is almost always to minimize run time. Most often, it’s big gains that matter - reducing run time from months or years to hours or days. Sometimes, however, researchers are willing to expend a great deal of money and/or effort to reduce run times by even a few percent.

Switching to a different language can and often does reduce run times by a factor of 100 or more. A little knowledge of how various languages function will help you determine which languages are appropriate for your situation. An overview is provided below.

16.4.2 Language or Religion?

Language and Operating System Evangelists

In the 1980’s, if you weren’t using C, you were a heretic.

In the 1990’s, if you were still using C, you were a heretic.

Since 2010, C has been making a comeback, despite not having changed significantly in decades.

C++ was the rage in application development for a while, then Java, then Scala, yada, yada.

Perl was the rage in server-side web programming until PHP and Ruby-on-rails came along. The saga continues.

There is no shortage of evangelists in the computer arena. Many of them will violently educate you on why their favorite language or operating system is “just better” and why people who use anything else are idiots.

Most such arguments take the form of "my language or OS has feature X and the others don’t", where "feature X" is something you never heard of and will probably never need.

Don’t pay attention to arguments that are obviously based on a bizarre emotional attachment to a piece of software. Those individuals are less interested in helping you make a good choice than they are in getting you to validate their choices and admire their cleverness.

The Rationalizations Behind Language Fever

Much of the language fervor is based on the common, but irrational, "newer is better" mind set.

The primary result of all this hype has not been significantly better programs, but a more fragmented computer industry where people have a harder time working together and companies have a harder time finding employees with specific language experience.

Billions of lines of C code are still in use today and rewriting all this code would be a foolish waste of man-hours. The same can be said for any other popular language.

While language choice can have a significant impact on performance, the programmer, not the language, determines the quality of the code in most other respects. Bad programmers will always find a way to write bad code no matter how well we design the language. Good programmers will write good code and be productive in old languages or new ones.

The Rational Approach

A programmer’s favorite language too often becomes a solution looking for problems. What we should be doing is assessing the needs of the program to be written and then choosing a language that best meets the needs.
The easiest language to use is usually the one you’ve practiced the most. If you practice a language that will work for most of your needs, then you will not find yourself in a conflict over using your favorite language or the most practical one for the job.

Base your choice of language on practical considerations: Portability, performance, availability, etc. Ultimately, this should be a sound economic decision. Which language will lead to the lowest overall cost for the project in the long term?

In order to calculate the long term costs, we have to understand a few things about programming languages. These things will be explained as you read on.

There is always a desire to use a language you already know and avoid the learning curve of using one that might be better suited for the job. This is a perfectly valid motive. There are hundreds of programming languages and nobody has time to learn the best one for each project they do. Just be sure to consider the benefits of learning a new language before you rule it out.

Using portable languages (languages that can be used on any operating system and hardware) will help minimize the number of languages you need to learn in order to be an effective programmer. You never know what operating systems or hardware your next program may need to run on.

You’ll want to master a compiled language, so that you can write fast code when you need to. All interpreted languages (i.e. scripting languages) such as Matlab, Perl, and Python have a finite list of built-in routines that they can perform efficiently. These routines are efficient because they are written in compiled languages, not the scripting language itself. If you have to implement your own algorithms that require a lot of computation, an interpreted language will be at least an order of magnitude slower than a compiled language.

If you avoid using a compiled language, you avoid developing skills that you will probably need someday to get results in time for a deadline. Practice using a compiled language regularly so that you’ll be able to get things done when you need to.

Coding in a compiled language will make your programs much faster but will not make them optimal. It will likely be at least an order of magnitude "less slow" than the interpreted version, though. Writing efficient code involves a variety of good practices, which are discussed in the appropriate context throughout this text.

All languages have limitations. The best we can do is choose languages with limitations we can accept. We can get around the inconvenience of doing certain things in C or Fortran that might be easier in a scripting language. We cannot get around the slow performance of interpreted languages. We simply have to switch to a compiled language in those situations.

Some advice for those of you who don’t want to spend much of your life learning computer languages:

• Master one popular, portable compiled language, so that you’re in a position to write fast code when you need to. For most people, this will mean learning C, C++, or Fortran.

• Master one scripting language for automating the execution of other programs. For most people, it makes very little difference which one, as long as it’s portable, i.e. you can use it on any operating system. Any of the Unix shells (Bourne shell, C shell, etc.) will do just fine, as will Lua, Perl, PHP, Python, Ruby, TCL, and a host of others.

If you master one portable compiled language and one portable scripting language, you’ll be in a good position to quickly write code for any purpose, which will mean a lot to you when you’re under pressure to make a deadline.

Object-Oriented Design (OOD) for Beginners

Computer geniuses who don’t understand the simple needs of many computational scientists will often push for object-oriented programming methods and languages.
Object-oriented design is a highly valuable practice and was the logical next step in the evolution of computer programming as programs became bigger and more complex.

However, there is widespread misunderstanding about what OOD is and how it applies to real problems.

Object-oriented design adds another level of organization to programs above subprograms. OOD groups subprograms together with the data structures they manage into classes. Like subprograms, classes are modules that can be developed and debugged separately and often reused in other programs.

This is primarily useful in very large programs with complex data structures. Most scientific programs are relatively small and use only simple data structures such as vectors and matrices. In these cases, OOD doesn’t generally offer much benefit, since subprograms alone provide all the modularity we need.

Second, OOD is a design methodology, not an implementation methodology. For this reason, software engineers generally prefer the term Object-oriented Design (OOD) over Object-oriented Programming (OOP). What this means is the object-oriented structure of a program is determined during the design phase, before you even decide what programming language to use.

Many inexperienced programmers think that choosing a so-called object-oriented language will result in an object-oriented program, but this is not at all the case. An object-oriented design can be implemented in any language and there are many programs written in object-oriented languages that do not follow object-oriented design methods at all.

Consider the C program and the Java program below. While C is not considered an object-oriented language, and Java is considered among the strictest of object-oriented languages, neither of these programs follows object-oriented design at all. Both use nothing but global variables to communicate between the functions (called methods in Java). The C program is a classic example of what we call spaghetti code, i.e. code without any structure. The Java program is spaghetti code wrapped in a class.

```c
#include <stdio.h>
#include <sysexits.h>

void sideEffect(void);

int global_variable1, global_variable2, global_variable3;

int main()
{
    global_variable1 = 1;
    global_variable2 = 2;
    sideEffect();
    printf("%d\n", global_variable3);
    return EX_OK;
}

void sideEffect()
{
    global_variable3 = global_variable1 + global_variable2;
    return;
}
```

```java
import java.io.*;
import java.util.Scanner;

class spaghetti
{
    static int global_variable1, global_variable2, global_variable3;

    public static void main(String args[])
    {
        global_variable1 = 1;
        global_variable2 = 2;
        sideEffect();
        System.out.printf("%d\n", global_variable3);
        return;
    }
}
```
public static void sideEffect()
{
    global_variable3 = global_variable1 + global_variable2;
    return;
}

Object-oriented languages simple provide features that make the object-oriented nature of the program more visible. They encourage the use of OOD, but do not cause it. As such, these language features amount to little more than a Jedi mind trick, saying "you don’t want to write spaghetti code, you want to go home and plan the structure of this program first”.

16.4.3 History of Programming Languages

Since the beginnings of computer science, there has been a drive to create better, easier programming languages, that would cause programmers to write better code with less effort.

The first push came mainly in the late 1950’s through the 1960’s and produced languages like COBOL (COmmon Business Oriented Language) and PL/I (Programming Language 1).

Efforts was largely focused on adding features to the language syntax to make it more intuitive.

The result was a phenomenon now known as creeping feature syndrome, where the number of features in software grows slowly and quietly, and before we realize it, has gotten out of control. This led to extremely complex languages and slow compilers that proved to have questionable benefit to code quality and programmer productivity, especially when weighed against the high costs associated with the compilers.

Around 1970, a team at AT&T led by Dennis Ritchie wiped the slate clean and created a new language from scratch, using the lessons of earlier languages.

The design philosophy was minimalist. Nothing would be part of the language syntax if it could be implemented reasonably well as a subprogram. This would keep the compiler simple and fast, in turn allowing it to spread rapidly to many different CPU architectures.

The result was the “C” language, which quickly became the most popular language in the world, and has since become the standard on which most newer languages (C++, Java, Perl, PHP, etc.) are based.

Instead of providing lots of built-in features, C provides only features with a high benefit/cost ratio and the ability to easily create and use collections of subprograms, called libraries. While the C language itself is minimal, the number of libraries available is vast and forever growing. There are free C libraries available for just about anything you might want to do with a computer. Finding a convenient way to implement a program in C is therefore mainly about finding the right libraries.

Ritchie and others embracing the minimalist philosophy noted that many language features, even rather basic ones such as support for character strings, are difficult to implement efficiently and provide little real benefit over functions that perform the same tasks.

For example, the ability to compare strings directly would add complexity to the compiler. If implemented in C, it would look like this:

```java
if ( string1 == string2 )
```

Ritchie’s team realized that string comparison could instead be implemented as a library function, which could actually be written in C. The syntax would not be as pretty, but it would not be a major inconvenience:

```java
if ( strcmp(string1,string2) == 0 )
```

By keeping this functionality out of the compiler, the compiler is kept simple, and these functions can be improved or replaced without altering the compiler.

Likewise, some languages provide syntactic support for basic matrix operations, such as:

```java
sum = mat1 + mat2
```
In C, this would be handled by a function call:

```c
matrix_add(mat1, mat2, sum);
```

This particular case illustrates another advantage of using a function call in some situations. The first example, \( \text{sum} = \text{mat1} + \text{mat2} \), is difficult to implement efficiently. In most languages, the expression on the right side of an assignment statement is evaluated first, the result stored in a temporary location, which is then assigned (copied) to \text{sum}. If the matrices are large, the need to allocate and then copy a temporary array can have a profound impact on the program’s memory requirements and run time, even causing the computer to run out of memory in some cases.

The \textit{matrix_add()} function, on the other hand, can easily store each result element directly in the \text{sum} array during the addition operation, without using any temporary space.

There are ways around the use of temporary arrays, but the reality is that languages providing this convenient syntax for matrices tend to require more memory and run slower.

Finally, there is a never ending debate about of programmer awareness vs abstraction. Much of language design is centered around abstracting complex operations, i.e. hiding the complexity from the programmer and making them appear simple.

Some would argue that it’s better for the programmer to be reminded about what’s really happening and that it may be computationally expensive for the computer. Hence making the programmer use functions like \textit{strcmp()} instead of making expensive operations like string comparison convenient and intuitive, will educate the programmer and discourage overuse of expensive operations in general, leading to more efficient code.

The bottom line is, don’t be fooled by hype about “newer, better” languages. In most cases, they provide little more than a prettier syntax, and often carry with them some serious disadvantages, which will become clear as you read on.

### 16.4.4 Compiled vs Interpreted Languages

#### Overview

When performance is a concern, it is highly advisable to use a purely compiled language.

Interpreted programs are run by another program called an interpreter. The most efficient interpreter spends more than 90% of its time parsing (interpreting) your code and less than 10% performing the useful computations it was designed for. Most spend more than 99% of their time parsing and less than 1% running. All of this wasted CPU time is incurred every time you run the program.

For program code that doesn’t do much computation, like interactive user interfaces, this isn’t necessarily a problem. For code doing intensive computation, however, using an interpreted language would be a travesty.

With a compiled program, the compiler does all this parsing ahead of time, before you run the program. You need only compile your program once, and can then run it as many times as you want.

Hence, compiled code tends to run anywhere from tens to thousands of time faster than interpreted code.

There is a middle class of languages, which we will call quasi-compiled for lack of a better term. The most popular among them are Java and Microsoft .NET languages. These languages are compiled to a byte code that looks more like machine language than the source code. However, this byte code is not the native machine language of the hardware, so an interpreter is still required to run it. Interpreting this byte code is far less expensive than interpreting the raw source code, so such programs run significantly faster than many other interpreted languages.

In addition to quasi-compilation, some languages such as Java include a Just-In-Time (JIT) compiler. The JIT compiler converts the byte code of a program to native machines language while the program executes. This actually makes the interpreted code even slower the first time it executes each statement, but it will then run at compiled speed for all subsequent iterations. Since most programs contain many loops, the net effect is program execution close to the speed of compiled languages.

Table 16.2 shows the actual run time (wall time) of a selection sort program written in various languages and run on a 3GHz Athlon64 under FreeBSD (outpaddling.cs.uwm.edu). Compilers and interpreters used included Clang 3.4.1, GCC 4.9, OpenJDK 1.6, Python 2.7, Perl 5.20, and R 3.2.2. GCC compilations were performed with standard optimizations (-O).

Run time was determined using the \texttt{time} command and memory use was determined by monitoring with \texttt{top}. 

---

Table 16.2 shows the actual run time (wall time) of a selection sort program written in various languages and run on a 3GHz Athlon64 under FreeBSD (outpaddling.cs.uwm.edu). Compilers and interpreters used included Clang 3.4.1, GCC 4.9, OpenJDK 1.6, Python 2.7, Perl 5.20, and R 3.2.2. GCC compilations were performed with standard optimizations (-O).

Run time was determined using the \texttt{time} command and memory use was determined by monitoring with \texttt{top}. 

---
Memory is allocated from a pool of virtual memory, which includes most RAM + a portion of disk known as swap. The first number shows the virtual memory allocated, and the second shows the portion of the allocated memory that resides in RAM, generally the most actively used blocks. Both numbers are important. How they actually impact the system and the performance of a program is complex and application-dependent.

This selection sort benchmark merely serves to provide a rough estimate of the relative speeds of languages when you use explicit loops and arrays.

Note that there are many different C, C++, and Fortran compilers available and the results will vary with different vendors’ compilers and even different versions of the compiler. Results will also vary with algorithms other than selection sort. However, the data here provide a pretty good representation of the relative speeds of programming languages.

<table>
<thead>
<tr>
<th>Language (Compiler)</th>
<th>Execution method</th>
<th>Time (seconds)</th>
<th>Memory</th>
</tr>
</thead>
<tbody>
<tr>
<td>C (clang 3.4.1 -O)</td>
<td>Compiled</td>
<td>5.85</td>
<td>14.5 MB (3.5 resident)</td>
</tr>
<tr>
<td>C++ (clang++ 3.4.1 -O)</td>
<td>Compiled</td>
<td>6.10</td>
<td>21.7 MB (3.5 resident)</td>
</tr>
<tr>
<td>C (gcc 4.9 -O)</td>
<td>Compiled</td>
<td>9.80</td>
<td>14.5 MB (2.6 resident)</td>
</tr>
<tr>
<td>C++ (g++ 4.9 -O)</td>
<td>Compiled</td>
<td>9.83</td>
<td>19.7 MB (3.5 resident)</td>
</tr>
<tr>
<td>Fortran (gfortran 4.9 -O)</td>
<td>Compiled</td>
<td>10.09</td>
<td>22.1 MB (3.5 resident)</td>
</tr>
<tr>
<td>Java 1.6 with JIT</td>
<td>Quasi-compiled</td>
<td>11.28</td>
<td>2,424 MB (33.4 resident)</td>
</tr>
<tr>
<td>Python 2.7 with numba JIT 0.29</td>
<td>Quasi-compiled</td>
<td>16.16</td>
<td>221 MB (77.5 resident)</td>
</tr>
<tr>
<td>GO 1.8</td>
<td>Compiled</td>
<td>25.51</td>
<td>11.3 MB (8.9 resident)</td>
</tr>
<tr>
<td>MATLAB</td>
<td>Mostly interpreted (limited JIT compilation)</td>
<td>120.86</td>
<td>763 MB (103 resident)</td>
</tr>
<tr>
<td>Java 1.6 without JIT</td>
<td>Quasi-compiled</td>
<td>361.51</td>
<td>2,424 MB (29.7 resident)</td>
</tr>
<tr>
<td>Python 2.7</td>
<td>Interpreted</td>
<td>1090</td>
<td>36.5 MB (16.5 resident)</td>
</tr>
<tr>
<td>Perl 5.20</td>
<td>Interpreted</td>
<td>2615 (43.58 minutes)</td>
<td>35.5 MB (14 resident)</td>
</tr>
<tr>
<td>R 3.2.2</td>
<td>Interpreted</td>
<td>17,747 (4.39 hours)</td>
<td>131 MB (41.8 resident)</td>
</tr>
<tr>
<td>C-shell</td>
<td>Interpreted</td>
<td>218,000 (extrapolated, 60.56 hours)</td>
<td>27.7 MB (9.9 resident)</td>
</tr>
</tbody>
</table>

Table 16.2: Selection Sort of 100,000 Integers

Not Everything in Interpreted Languages is Interpreted

Many interpreted languages also offer a variety of built-in functions or libraries that execute at compiled speed (because they are written in compiled languages). One example is the NumPy library available for Python, a popular interpreted language. It contains subprograms for mathematical computation written mostly in C, C++, and Fortran.

If your Python program does most of its computation using these compiled functions, it will run nearly as fast as a compiled program. However, if you rely on such features, you are limited to the finite set of built-in capabilities provided by the language or the compiled libraries such as numpy. Not every array and matrix operation can be accomplished using built-in functions or vector operations, and operations implemented in an interpreted language will be extremely slow.

Using a Compiled Language Doesn’t Guarantee a Fast Program

Choosing a compiled language is very important to execution speed, but is not the whole story. It will not guarantee that your code is as fast as it could be, although it will be far less slow than the same algorithm implemented in an interpreted language.

Choosing the best algorithms can be actually far more important in some cases. For this reason, computer science departments focus almost exclusively on algorithms when teaching software performance.

Suppose, for example, you have a selection sort written in C and a heap sort written in Python. There are a few factors that will determine execution time:

- Selection sort is an \( O(N^2) \) algorithm, which means that the execution time of the selection sort in C is proportional to \( N^2 \) for a list of \( N \) elements. I.e., the run time is \( N^2 \) times some constant \( K_{\text{C-selection}} \).

\[ \text{time} = K_{\text{C-selection}} \times N^2 \]
• Heap sort is an \(O(N \times \log(N))\) algorithm, which means that the execution time of the heap sort in Python is proportional to \(N \times \log(N)\) for a list of \(N\) elements:

\[
time = K_{\text{Python-heap}} \times N \times \log(N)
\]

The difference in speed between C and Python is essentially constant, while \(N^2\) grows faster than \(N \times \log(N)\) as \(N\) increases. Hence, there exists a value of \(N\) above which the selection sort will take longer, regardless of the values of the constants. 

\(K_{\text{C-selection}}\) and \(K_{\text{Python-heap}}\) can be measured directly by running each program with a reasonably large list size. (A very small list size may allow program overhead to dominate the run time.) Suppose we find that \(K_{\text{Python-heap}} = 100 \times K_{\text{C-selection}}\). This would be reasonable given that C is about 100 times faster than Python according to Table 16.2. In this case, we can estimate the value of \(N\) above which the C selection sort will take longer by solving the following inequality:

\[
K_{\text{C-selection}} \times N^2 > K_{\text{Python-heap}} \times N \times \log(N)
\]

\[
K_{\text{C-selection}} \times N^2 > 100 \times K_{\text{C-selection}} \times N \times \log(N)
\]

\[
N^2 > 100 \times N \times \log(N)
\]

\[
N > 100 \times \log(N)
\]

This inequality is satisfied by a value of 238, assuming we use base 10 log throughout (\(\log(238) = 2.37\) and 238 > 100 * 2.37). Of course, the best solution here is to write a heap sort in a compiled language and the worst is a selection sort in an interpreted language.

There are also cases where choosing the right algorithm can make less significant, but measurable differences. For example, there are a variety of algorithms for solving systems of equations. Some of them work best on sparse matrices, others on triangular matrices, and others on dense matrices. Some languages and libraries offer high-level functions that will automatically try to choose the best algorithm for you, but it’s always best to understand what your program is doing and be prepared to choose the best algorithms yourself.

**Think About the Future**

You may find that an interpreted language is fast enough for most of your work, but eventually you may want to run bigger cases and find that your programming effort was in vain and you now have to rewrite your code in a compiled language. This is a very common occurrence in research computing.

Many interpreted language programs have been completely or partially rewritten in compiled languages in order to improve performance. This is an unfortunately duplication of effort that could have been avoided with some forethought.

**Compiled Languages are Not That Hard to Use**

You may encounter criticism of compiled languages for being harder to use because they lack "built-in" functions such as common vector and matrix operations, etc. These criticisms are often misleading. Those who circulate them are often unaware that these functions are readily available in free libraries, so their difficulty stems from unnecessarily reinventing the wheel. It doesn’t really matter whether a function is formally part of the language if it is easy to acquire and use.

For example, most common vector and matrix operations are readily available to C, C++, and Fortran programs in free libraries such as BLAS, LAPACK, Eigen, and many others. These libraries can be installed in seconds on most modern Unix systems using a package manager. To install LAPACK on FreeBSD, for example, simply run the following command as root:

```
pkg install lapack
```

In fact, these same libraries provide much of the functionality in interpreted languages like MATLAB and Python.

It is often (erroneously) stated that interpreted languages are easier to use because you don’t have to compile the program before you run it. However, programs need only be compiled once and can then be used indefinitely without recompiling. Compilation on modern computers is also very fast. A typical researcher’s scientific program will compile in a few seconds, so even if you’re making and testing changes to the code, compilation will not be an inconvenience.

Some might be tempted to argue "So what if it’s slower than a compiled program: I can just use more processors." However, using an expensive resource like a cluster to make slow programs run faster is unethical, as it wastes costly resources that could be better utilized by more efficient programs. An interpreted program could be hundreds of times slower than its compiled equivalent and it would not be cost effective to solve performance issues with hundreds of times more hardware. The software itself must be made as fast as possible before we consider throwing more hardware at it.
Memory Use

Note also that when you run an interpreted program, the interpreter is competing for memory while the program is running. Hence, in addition to running an order of magnitude or more slower than a compiled program, an interpreted program will generally require more memory resources to accommodate both your program and the interpreter at the same time.

16.4.5 Portable Software

Portable software is software that can be used on many different operating systems and hardware platforms. Like anything else that’s portable, you can take it with you to any computer you may want to run it on.

Examples of portable languages include C, C++, Fortran, Pascal, Perl, and Python, to name a few. Portable programs include the Firefox web browser, LibreOffice and OpenOffice office suites, etc.

Non-portable software can only be run on one or a few operating systems and hardware platforms. Examples of non-portable languages include Visual Basic and C#, which are available only on Microsoft Windows, and AppleScript, which is only available on Apple operating systems. Some non-portable software applications include Microsoft Office, which runs only on Microsoft Windows and Mac. Most commercial software, in fact, runs only on Microsoft Windows and Mac. Some runs only on one or the other, and a small number of commercial software products run on one or a few specific Unix systems such as Linux, Solaris, or IBM AIX.

Since learning to use any program or language is work, using portable software will save you a lot of time and effort when you find a need to run programs on a different platform. This situation is inevitable for almost everyone, so if you’re thinking ”I’ll just stay with the operating system I’m used to”, guess again. The day will come when you’re forced to use a different system, so it’s best to start preparing now. By using portable software, you will develop portable skills, and you won’t have to waste time and effort learning new software and languages in order to run your programs on other computers.

16.4.6 Common Compiled Languages

C/C++

C and C++ are the most popular compiled languages for general use over the past several decades. Both are commonly used for scientific programming.

Most compilers and interpreters are written in C and C++.

C is a very simple, but powerful high level language that was developed in unison with the Unix operating system. In fact, most of the Unix operating system is written in C. C became the dominant general-purpose programming language on other operating systems such as DOS and Windows during the 1980’s. DOS and Windows are also largely written in C. C is best known for producing the fastest code possible in a portable high-level language.

C++ is an extension of C that adds a wealth of new features, many of which are aimed at object-oriented programming. Object-oriented programming encourages more modular design of programs, which can be critical for large applications. Most modern languages now have features to support object-oriented programming.

Unlike C, C++ is a complex language that requires a great deal of knowledge and experience in order to use effectively. It is a powerful tool for scientific programming, but has a much high learning curve than C. Mastering the C++ language and keeping up with the new features that are frequently added is a career in and of itself.

Fortran

Fortran (Formula Translator) was the first major high-level language, created by a team at IBM led by John Backus in the 1950’s. Fortran is a compiled language, originally designed for scientific computing, and has been the most popular scientific programming language throughout most of computing history. Fortran has always had many convenient features for scientific computing, such as built-in mathematical functions, and seamless, efficient support for complex numbers. Performance of Fortran programs is comparable to C or C++.

Fortran has gone through several major evolutionary steps. Fortran 77 was the standard scientific language for quite some time. Fortran versions up to and including Fortran 77 were often criticized for being cumbersome and limited in comparison to more
modern languages. Because of these limitations, Fortran lost some market share to C, C++, and specialized languages such as MATLAB.

The Fortran 90 standard added many powerful capabilities and eliminated most of the more antiquated limitations, such as strict column formatting (a vestige of the days of punch-cards). Fortran 90, and the more recent Fortran 2003 and 2008, are powerful languages with many modern features, including object-oriented support (for those who are interested).

Fortran is still the language of choice in many areas of scientific computing, including weather forecasting, molecular modeling, etc. A great deal of Fortran software is under active development today, and the future of Fortran appears to be bright.

Newer Fortran versions include many features that are of limited value to most programmers.

Conclusion

For the average scientist of engineer who needs to write fast code, C (not C++) or Fortran 90 are probably the simplest choices. They are fairly simple languages that can be mastered in a relatively short time, unlike C++ and later Fortran standards. They also provide all the features that are needed by most scientific programmers, who do not write very large programs full of complex data structures.

16.4.7 Common Interpreted Languages

MATLAB and Octave

MATLAB is an interpreted scripting language designed specifically for vector and matrix calculations common in science and engineering. MATLAB handles vectors and matrices with an intuitive syntax, offers a wide variety of built-in, pre-compiled routines for numerical computations, and has other useful features such as sophisticated graphing and plotting routines.

Caution

MATLAB is not a general-purpose programming language and is not substitute for high-performance, general-purpose programming languages like C, C++, and Fortran. It is a useful tool to learn in addition to general-purpose programming languages, but is limited in what it can do efficiently. MATLAB's built-in functions for matrix operations such as solving linear systems are about as fast as similar functions used in C, C++, or Fortran. In fact, MATLAB often uses the same functions found in C, C++, and Fortran libraries such as the BLAS, LAPACK, UMFPACK, etc. Complex calculations coded in MATLAB's scripting language, however, will generally be unacceptably slow.

If used properly, and for its intended purpose (vector and matrix calculations), MATLAB can perform as well as compiled languages in some cases. Proper use of MATLAB means utilizing its many built-in matrix capabilities and routines (vectorizing the code), which use the pre-compiled routines of the MATLAB interpreter. (MATLAB itself is written in C, C++, and Fortran.)

Unfortunately, not all calculations can be done efficiently in MATLAB. Some operations are difficult or impossible to vectorize, and MATLAB has a finite number of built-in functions. Even if the code can be vectorized, doing so is often more difficult that writing the code in a compiled language.

Many users use MATLAB where it is ill-suited for the simple reason that it is the language they are most familiar with. MATLAB programs that use explicit loops to process data one element at a time will run orders of magnitude slower than equivalent compiled programs or a MATLAB program utilizing MATLAB’s vector capabilities and/or pre-compiled functions.

To illustrate this point, the following program shows three ways of accomplishing the same basic vector initialization in MATLAB. The first uses the MATLAB colon operator to iterate through values from 1 to 100,000. The second uses the pre-compiled linspace() function (which is probably written in C) to accomplish the same iteration. The third uses an explicit MATLAB loop, which must be performed by the MATLAB interpreter. Each toc statement reports the time elapsed since the previous tic statement.
fprintf('Colon operator:
');
tic
list1 = [1:100000];
toc

fprintf('linspace function:
');
tic
list2 = linspace(1,100000);  
toc

fprintf('Explicit loop:
');
tic
for c = 1:100000
    list3(c) = c;
end
toc

Below is the output produced by MATLAB running on a 2GHz Core Duo system:

Colon operator:
Elapsed time is 0.002485 seconds.
linspace function:
Elapsed time is 0.008148 seconds.
Explicit loop:
Elapsed time is 81.661672 seconds.

Below is the output produced by Octave running on the same Core Duo system:

Colon operator:
Elapsed time is 0.0000169277 seconds.
linspace function:
Elapsed time is 0.000270128 seconds.
Explicit loop:
Elapsed time is 0.994528 seconds.

All three methods accomplish the exact same result, but the colon operator and linspace function do it in a small fraction of the time taken by the explicit loop. This is due to the fact that the interpreter need only parse one statement, one time, for the colon operator and linspace() function methods, whereas it must parse three statements 100,000 times each for the explicit loop. The linspace() function probably takes somewhat longer to to the overhead of calling a function.

Note that all three methods do in fact use loops, but the first two use compiled loops while the third uses an interpreted loop. Note also that C code below executes in about 0.002 seconds, comparable to the MATLAB code without an explicit loop.

```c
size_t c;
double list[LIST_SIZE];
for (c=0; c < 100000; ++c)
    list[c] = c;
```

This example illustrates that if MATLAB has no built-in feature such as the vector operation or pre-compiled routine such as linspace() to perform the computations you require, you will need to write code in a compiled language in order to achieve good performance.

MATLAB provides tools for integrating C and Fortran code into MATLAB programs for this purpose. Mixing programming languages in this way is complex, however, so it may be easier to simply write the entire program in a compiled language.

If you’re going to write a sophisticated application, you may be better off writing the entire program in a compiled language. It’s better to get the practice with the compiled language from the beginning than to have to learn it in the 11th hour of your project as well as learn how to integrate compiled code with MATLAB.

Even if used properly, MATLAB may still be limiting compared with compiled languages. Because of the way MATLAB handles matrices, they are usually entirely in memory during processing. MATLAB is notorious for using far more memory than other
languages. MATLAB’s high memory use can impact performance in some cases, and in other cases even prevent programs from completing on systems with insufficient memory.

One more potential barrier, especially in parallel computing, is the fact that MATLAB is an expensive commercial product and requires a license server in order to run on a cluster.

The need to allocate funds, procure licenses, and set up a license server inevitably delays results. In addition, you will not be able to run MATLAB while the license server or the network is down.

One possible solution to this issue is Octave, a free and open-source tool very similar to MATLAB.

The Octave interpreter is nearly 100% compatible with MATLAB. In fact, the Octave project considers any differences to be bugs.

The main limitation of Octave is that it does not provide full compatibility with the less common MATLAB toolboxes. However, most common MATLAB code and many of the more common toolbox functions (e.g. from the statistics toolbox) will run under Octave without modification.

Another important difference is that Octave graphical capabilities are not fully compatible with MATLAB. The most common plotting and graphing functions are the same, but duplicating all the powerful graphical features of MATLAB is a huge task. This is, of course, irrelevant on a cluster and only a concern for desktop and laptop users. Many users use Octave to do their parallel computations and MATLAB on their personal computers for visualizing the results.

Starting with Octave 3.8.0, there is a new graphical user interface (GUI) based on the QT cross-platform GUI tools. The Octave GUI is very similar to MATLAB’s GUI and faster. (The Octave GUI is written in C++, whereas MATLAB’s is written in Java.)

The Octave GUI

The general consensus at the time of this writing is that MATLAB and Octave perform about the same for properly vectorized code, while MATLAB performs better on “bad” code (using explicit loops). Octave is faster at some algorithms and MATLAB
is faster at others. In fact, MATLAB and Octave utilize many of the same open source C, C++, and Fortran libraries for many of their common functions.

Below is a comparison of a script run under both Octave 3.4.3 and MATLAB R2012a. As you can see, some sections of the program ran faster under Octave and others ran faster under MATLAB.

<table>
<thead>
<tr>
<th>Octave 3.4.3</th>
<th>MATLAB R2012a</th>
</tr>
</thead>
<tbody>
<tr>
<td>Creating nodes...</td>
<td>Creating nodes...</td>
</tr>
<tr>
<td>Elapsed time is 0.1856 seconds.</td>
<td>Elapsed time is 0.019867 seconds.</td>
</tr>
<tr>
<td>Creating T1...</td>
<td>Creating T1...</td>
</tr>
<tr>
<td>Elapsed time is 0.002146 seconds.</td>
<td>Elapsed time is 0.000020 seconds.</td>
</tr>
<tr>
<td>Creating 2D discrete operator...</td>
<td>Creating 2D discrete operator...</td>
</tr>
<tr>
<td>Elapsed time is 75.51 seconds.</td>
<td>Elapsed time is 60.261948 seconds.</td>
</tr>
<tr>
<td>Size of B matrix: 9604 x 9604</td>
<td>Size of B matrix: 9604 x 9604</td>
</tr>
<tr>
<td>Calculating initial condition...</td>
<td>Calculating initial condition...</td>
</tr>
<tr>
<td>Elapsed time is 0.3859 seconds.</td>
<td>Elapsed time is 0.014246 seconds.</td>
</tr>
<tr>
<td>Calculating M...</td>
<td>Calculating M...</td>
</tr>
<tr>
<td>Elapsed time is 3.303 seconds.</td>
<td>Elapsed time is 3.366316 seconds.</td>
</tr>
<tr>
<td>Starting main loop...</td>
<td>Starting main loop...</td>
</tr>
<tr>
<td>Elapsed time is 10.24 seconds.</td>
<td>Elapsed time is 12.080297 seconds.</td>
</tr>
</tbody>
</table>

A Real Example for Comparison

Now let’s consider a simple real-world example, namely approximating the area under a curve. There are a variety of approaches, such as the trapezoid rule. This example uses rectangular slices with the height equal to the function value at the center of the slice, which provides a good estimate and is very easy to program.

Each of the following examples was run using 10,000,000 slices.

The first example program illustrates a typical approach to solving the problem in MATLAB:

```matlab
% Program description:
% Compute the area under the curve if the square root function.
% Demonstrates the value of MATLAB’s just-in-time compiler.
% Unlike the more complex selection sort example, MATLAB is able
to compile the explicit loop below at run-time, making it almost
as fast as a vector operation.
% History:
% 2013-07-04 Jason Bacon Begin

tic
first_x = 0;
last_x = 1;
slices = 1000000;
slice_width = (last_x - first_x) / slices;

x = first_x + slice_width / 2;
estimated_area = 0;
while x < last_x
    slice_area = sqrt(x) * slice_width;
estimated_area = estimated_area + slice_area;
x = x + slice_width;
end
fprintf('Final x = %0.16f Estimated area = %0.16f\n', x - slice_width, estimated_area);
```

```matlab
```

```
toc

Final x = 0.9999999497501700  Estimated area = 0.6666666666390187
Elapsed time is 0.538943 seconds.

The example above uses an explicit loop, which could be too slow for a large number of slices, because the MATLAB interpreter has to execute the statements in the loop many times. Below is a vectorized version of the same program:

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Program description:
% %
% % Demonstrates the use of vector operations. Vector operations are
% % generally much faster than explicit loops, but require much more
% % memory in cases where vectors or matrices are not otherwise needed.
% % This may cause the program to actually run slower than one without
% % vectors, or even fail if the vectors used are too large to fit in
% % memory.
% %
% % History:
% % 2013-07-04 Jason Bacon Begin
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

tic
first_x = 0;
last_x = 1;
slices = 10000000;
slice_width = (last_x - first_x) / slices;

x = linspace(first_x + slice_width / 2, last_x - slice_width / 2, slices);
slice_area = sqrt(x) * slice_width;
estimated_area = sum(slice_area);
fprintf('Final x = %0.16f Estimated area = %0.16f\n', x(slices), estimated_area);
toc

Final x = 0.9999999500000000  Estimated area = 0.6666666666685899
Elapsed time is 0.434779 seconds.

The vectorized version uses no explicit loops, so the MATLAB interpreter will not be a bottleneck.

Interestingly, however, it does not run much faster than the non-vectorized code. This is partly because the program is so simple that MATLAB’s just-in-time (JIT) compiler can compile the explicit loop for us. More complex programs will not benefit from the JIT compiler as much, however, so explicit loops should be avoided where speed is essential. Speed is also limited by the increase in memory use due to the use of large arrays of 10,000,000 elements. This memory use overwhelms the CPU cache, forcing the program to use slower main memory.

While using vector operations will generally speed up MATLAB code immensely, the down side is generally an increase in memory use. If we were to increase the number of slices used by this program to 1 billion, the program would now use two vectors of 8 billion bytes (8 gigabytes) each. (Each value in MATLAB program typically requires 8 bytes of memory). If your computer does not have 16 gigabytes of available RAM, the program simply won’t run. Even if it does, it will not be able to utilize the CPU cache very well, and will take much more time per slice than it does for smaller slice counts.

When programming in MATLAB, we are often faced with a choice between using explicit loops, which are very slow, and using large amounts of memory, which also slows down the program and potentially prevents it from running due to memory exhaustion.

A C version of the program is shown below:
/* Description:
 * Compute the area under the curve if the square root function.
 * 
 * Arguments:
 * None.
 * 
 * Returns:
 * Standard codes defined by sysexit.h
 * 
 * History:
 * Date Name Modification
 * 2013-07-04 Jason Bacon Begin
 */

#include <stdio.h>
#include <math.h>
#include <sysexits.h>

int main(int argc, char *argv[])
{
    double first_x, last_x, x, slice_width, estimated_area;
    unsigned long slices;
    first_x = 0.0;
    last_x = 1.0;
    slices = 10000000;
    slice_width = (last_x - first_x) / slices;
    // Use x in the middle of each slice
    x = first_x + slice_width / 2;
    estimated_area = 0.0;
    while (x < last_x)
    {
        estimated_area += sqrt(x) * slice_width;
        // Fast, but causes x to drift due to accumulated round off when
        // adding imperfect x + imperfect slice_width
        x = x + slice_width;
    }
    printf("Final x = %0.16f Estimated area = %0.16f\n", x - slice_width, estimated_area);
    return EX_OK;
}

Final x = 0.9999999497501700 Estimated area = 0.6666666666390187
0.11 real 0.10 user 0.00 sys

Sine C is a compiled language, there is no disadvantage to using explicit loops, and no need to use arrays. As a result, this program is much faster than either MATLAB script, and will never be limited by available memory.

Note that this program, as well as the MATLAB program with an explicit loop, has a potential issue with drift caused by round-off error, which is inherent in all computer floating point values. (Floating point is a storage format similar to scientific notation, which is used by computers to approximate the real number set.)
If the width of a slice cannot be represented perfectly in the computer’s floating point format, then each successive value of x will be off by a little more than the previous. For example, representing 1/10 in computer floating point is like trying to represent 1/3 in decimal. It requires an infinite number of digits to represent accurately, so a computer can only approximate it to about 16 decimal digits.

After a large number of slices are processed, the value of x could drift far from the center of the theoretical slice we were aiming for. The MATLAB linspace() function, used in the vectorized example, avoids this issue. The C program below demonstrates how to avoid this problem by basing each value of x on the initial value, instead of the previous value:

```c
#include <stdio.h>
#include <math.h>
#include <sysexits.h>

int main(int argc, char *argv[])
{
    double first_x, last_x, base_x, x, slice_width, estimated_area;
    unsigned long slices, current_slice;
    first_x = 0.0;
    last_x = 1.0;
    slices = 10000000;
    slice_width = (last_x - first_x) / slices;
    current_slice = 0;
    estimated_area = 0.0;
    base_x = first_x + slice_width / 2; // Start in middle of first slice
    while (current_slice <= slices)
    {
        // Compute each x directly from a base value to avoid accumulating
        // round-off error. This is slightly slower than x = x + slice_width
        // due to the multiplication, but produces more accurate results.
        x = base_x + (slice_width * current_slice++);
        estimated_area += sqrt(x) / slice_width;
    }
    printf("Final x = %0.16f Estimated area = %0.16f
", x - slice_width, estimated_area);
    return EX_OK;
}
```
This version produces a final value of x that is exactly what it should be.

<table>
<thead>
<tr>
<th>Final x</th>
<th>Estimated area</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.9999999500</td>
<td>0.6666667666686176</td>
</tr>
</tbody>
</table>

0.11 real 0.11 user 0.00 sys

Because this version of the program uses an additional multiply operation to compute each value of x, it is slightly slower. It does, however, produce more accurate results, especially for high slice counts.

**R**

R is an open source statistics package with a built-in scripting language. It is a very popular substitute for commercial systems such as SPSS and SAS.

Like Matlab/Octave, the R scripting language is a vector based language and can perform many common vector operations quickly, because the R interpreter uses compiled loops behind the scenes.

Explicit loops in R scripts are interpreted and extremely slow, and seen in Table 16.2.

Like Matlab/Octave, R code should therefore be vectorized as much as possible by replacing explicit loops with vector operations or calls to compiled functions.

Since the only way to achieve good performance is by vectorizing code, R is also prone to use far more memory than an equivalent program written in a general purpose language. However, if used appropriately, R has its place in high performance computing.

The R CRAN project provides add-on R packages that can be easily installed from within R by any user. Many of these packages integrate compiled C, C++, or Fortran code in order to maximize speed and flexibility.

**Perl**

Perl is a part shell, part programming language. It is an interpreted language with many of the same convenient syntactic features of the Unix shells, plus a wealth of built-in functions such as you would find in the standard C libraries. Perl is designed primarily for text-processing, and is commonly used on WEB servers to process data from WEB forms. However, Perl has been successfully used for a wide range of applications, and is popular in scientific computing for textual data processing where computational performance is not critical.

**Python**

Python is generally regarded as a scripting language, but is popular enough in scientific programming to be mentioned here. Compiled libraries such as scipy (sigh-pie) and numpy (num-pie) make it possible to write efficient scientific programs in Python, with the same caveats as MATLAB or Octave. As long as most of the calculations are performed inside the compiled library routines, performance should be acceptable. Iterative code written in Python to do intensive computation, however, will result in run times orders of magnitude longer than equivalent C, C++, or Fortran code.

**Mixing Interpreted and Compiled Languages**

Some developers write much of their code in an interpreted scripting language and only the portions that require speed in a compiled language.

If you are using Unix shell scripts, there is no integration necessary. You simply write complete programs in the compiled language and run them from your scripts.

There are also tools available for integrating compiled subprograms into many interpreted languages.

The MEX system allows you to write subprograms in C, C++, or Fortran and integrate them into a MATLAB script. Octave also supports MEX as well as its own system MKOCTFILE.
Cython is a Python-to-C translator and module generator. It allows you to write code in an extended Python syntax and compile it for better performance. The Cython compiler generates C or C++ source code that can be compiled to a Python module, which can be imported into Python programs. You may not get the same performance you would from hand-written C, but it will be orders of magnitude faster than interpreted Python.

There are also tools such as SWIG, which generate interfaces to compiled code for a variety of interpreted languages.

It’s important to consider whether you want to deal with the complexity of integrating multiple languages. Savvy programmers developing complex applications may prefer this approach while less savvy users or those writing simpler programs may find it easier write the entire program in C or Fortran.

16.4.8 Summary

The pool of different programming languages in existence today is insurmountable and growing at an accelerating rate. As a result, it is becoming harder and harder to find programmers with skills in a specific language.

The best approach to learning is to develop good general programming skills that will apply to most languages. Stay away from proprietary or esoteric language constructs, tools, and techniques, and focus on practices that will be useful in many languages and on many operating systems.

To develop depth of knowledge, focus on mastering one general-purpose scripting language and one general-purpose compiled language. The skills you develop this way are more likely to serve you directly, but will also help you conquer other languages if it proves necessary.

Don’t be taken in by the false promises of "higher level" languages. The reality is that programming doesn’t get much easier than it is in C or Fortran. Other languages may seem easier to the beginning programmer, but as you progress you will discover their limitations.

16.4.9 Self-test

1. What does a compiler do?
2. What does an interpreter do?
3. Which offers better performance, compiled languages or interpreted languages? Why? How much of a difference is there?
4. If a MATLAB script using mainly explicit loops to solve a matrix takes 10 hours to run, roughly how long would the same program written in Fortran take? Why? How accurate would you expect your estimate to be? Why?
5. If a MATLAB script using mainly vector operations and built-in functions to solve a matrix takes 10 hours to run, roughly how long would the same program written in Fortran take? Why? How accurate would you expect your estimate to be? Why?
6. What is an open standard? Is it the same as open source?
7. What is the main advantage of using open-standard languages?
8. What is the best language for developing high-performance computing applications and libraries? Explain.
9. Can you use two different C compilers to build components of the same program? Explain.
10. Is it possible to mix C, C++, and Fortran code in building a single program? Explain.
11. What must be done to achieve reasonable performance in MATLAB and Octave programs?
12. Describe two major advantages of a compiled language over an interpreted language.
13. What is a possible advantage of interpreted languages over compiled languages?
14. Describe what happens when a compiled program is executed.
15. Describe what happens when an interpreted program is executed.
16. Rank the following languages in order of program execution speed.

- C
- C-shell
- C++
- Java (with JIT)
- Perl

17. What is the JIT compiler, and what does it do?

16.5 Engineering Product Life Cycle

Introduce now before developing bad habits. Should not wait until software engineering class and change the way we think.

This section introduces the product life cycle, which is used to assure quality in all fields of engineering, including software engineering. In software engineering, we refer to it as the software life cycle. Although the software life cycle may not be the primary focus of this course, it should be practiced in all programming endeavors, including college courses, personal projects, and professional development.

The product life cycle has been extensively studied and refined over time, and is the topic of entire semester courses in most engineering disciplines. Our coverage here is a very brief overview, using a 4-step process which is outlined in the following sections.

16.5.1 Specification

Specification is understanding the essence of the problem to be solved in as much detail as possible. Specifications may evolve during design and implementation stages as new insights are gained from working on the solution. However, every effort should be made to write specifications that will require minimal change during later stages of development.

16.5.2 Design

The design phase involves examining possible solutions to the problem with a completely open mind. The decision to write software or otherwise develop a new solution does not occur until after the design phase. Instead, the design phase focuses on the process of solving the problem. This process could be carried out by hand, by existing equipment or technology, by a software solution, or a new hardware solution.

In the design stage, an engineer may develop flowcharts, equations, top-down designs, drawings, etc., but not program code or hardware prototypes. One should avoid any thoughts about how the solution will be implemented during the design phase. Such thoughts are a distraction from the purpose here, which is to develop theoretical solutions.

Failure to fully explore solutions in the design phase can lead to reinventing a wheel that does not need reinventing, or more commonly, unnecessary difficulties in the implementation stage. Developing a well-thought-out design is the best investment of time you can make in the development of a product. Each hour invested in developing a good design could save 10 or 100 hours of aggravation during implementation.

16.5.3 Implementation

Implementation involves building something to test the process developed in the design stage. If you developed a good design, the implementation stage will be relatively straightforward and uneventful. If you find yourself struggling during implementation, then you need to go back and correct deficiencies in the design.

If the best solution found during the design stage is to use existing hardware or software, there is little to do in this stage. If it involves developing new hardware or software, then implementation involves the following:
Selecting the right tools and materials. For software, this means computer hardware, operating system, and programming language. For a hardware design, it means electronic or mechanical devices and fabrication techniques. A good choice here requires a solid understanding of the design, and knowledge of many available tools. Far too often, software developers choose an operating system or language because it’s the only one they know, leading to a poor quality product that does not serve the customers’ needs very well.

Performing the implementation. For software, this means writing the code. For hardware, it could mean building prototypes of the hardware or programming an FPGA.

During implementation, you may discover a need to alter the design or specification due to unforeseen difficulties with implementation. However, a good specification and design will require only minimal changes.

16.5.4 Testing

Testing is the heart of good engineering. Solid scientific theories and technology can help us design and build products faster and cheaper, but testing is the only way to ensure quality.

Testing is not a separate stage in the development timeline, but occurs continuously starting at the beginning of the implementation stage, and continues indefinitely, long after implementation and product release.

Testing begins the moment there is something to test, and never ceases as long as the product remains available.

Testing should occur following each small change throughout the implementation process. This type of testing is known as integration testing, since each new piece is tested the moment it is integrated into the product. In software development, integration testing occurs following each small change to the code, such as adding a new loop, if block, or subprogram stub.

Additional types of testing occur following completion of the product, such as alpha testing, which refers to in-house testing of the complete product before releasing it to customers, and beta testing, which refers to testing performed by a limited group of customers before officially releasing the product for general use.

Integration testing should catch 99% of the bugs in a program. Alpha testing should catch almost all of the few remaining bugs before the program is released for beta testing.

Alpha testing follows implementation, and should use a significant fraction of the total project time (typically 1/4 to 1/2, depending on the reliability requirements). If the project duration is six months, you should aim to complete implementation in roughly four months to allow for thorough alpha testing and bug fixes. Bugs found after implementation is complete tend to be difficult to identify and resolve, so you must allocate a generous amount of time for alpha testing.

Customers should never be the ones to find bugs in your code! This is bad for your reputation as a programmer or company. Beta testing should be viewed as an opportunity to showcase a new product and demonstrate the quality of your development process, not as an opportunity to get free testing from end-users.

Note

Every script or program should be tested on more than one platform (e.g. BSD, Cygwin, Linux, Mac OS X, etc.) immediately, in order to shake out bugs before they cause problems.

The fact that a program works fine on one operating system and CPU does not mean that it’s free of bugs. By testing it on other operating systems, other hardware types, and with other compilers or interpreters, you will usually expose bugs that will seem obvious in hindsight.

As a result, the software will be more likely to work properly when time is critical, such as when there is an imminent deadline approaching and no time to start over from the beginning after fixing bugs. Encountering software bugs at times like these is very stressful and usually easily avoided by testing the code on multiple platforms in advance.

16.5.5 Self-test

1. Describe each of the stages of the software life cycle.

2. How important are the specification and design stages? Why?
3. Who is responsible for specifications? Why?
4. What should be decided in the design stage?
5. Describe the three major types of testing discussed in this chapter. What role does each play in the life cycle?
Chapter 17

Data Representation

17.1 What Everyone Should Know About Data Representation

Computer science students typically spend at least one or two semesters studying computer architecture (hardware design) and machine language. The purpose of this training is not to prepare them for careers as hardware designers or machine language programmers, but to make them better programmers through an understanding of computer hardware.

Knowing the limitations of computer number systems is necessary in order to write programs that produce correct and precise output, and also in writing the most efficient programs possible.

This chapter provides a very brief overview of these limitations for non-computer scientists.

17.2 Orders of Magnitude

Working with computers requires knowing some terminology referring to quantities of data. The most commonly used terms and prefixes are outlined in Table 17.1 and Table 17.2.

<table>
<thead>
<tr>
<th>Term</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>bit</td>
<td>Binary Digit</td>
</tr>
<tr>
<td>byte</td>
<td>8 bits</td>
</tr>
<tr>
<td>nybble</td>
<td>4 bits</td>
</tr>
</tbody>
</table>

Table 17.1: Basic Quantities

Prefixes exist for both metric (decimal) and binary measurements. The binary prefixes use the same first two letters as the metric, followed by ‘bi’ (from binary).

Metric and binary prefixes are often confused and interchanged. This is not a problem where accuracy is not critical, such as in casual conversation, since they represent very similar values. For example, it is often stated that a computer has 4 gigabytes (4 x 10^9 bytes) of RAM, when in fact it has 4 gibibytes (4 x 2^30 bytes).

17.3 Numeric Limitations of Computers

Computers cannot represent infinite mathematical sets like the integers or reals. Each digit in a number requires memory in which to store it. An infinitely large number has an infinite number of digits, and no computer has infinite memory for storing all these digits.

In addition to the limits on memory, computer CPUs have much smaller limits on how much data they can process at once. This limit is known as the CPU’s word size. At the time of this writing, most computers can process at most 64 bits at once, which
Table 17.2: Prefixes for larger quantities

<table>
<thead>
<tr>
<th>Prefix</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>kilo-</td>
<td>10^3 (1,000)</td>
</tr>
<tr>
<td>kibi-</td>
<td>2^10 (1,024)</td>
</tr>
<tr>
<td>mega-</td>
<td>10^6 (1,000,000)</td>
</tr>
<tr>
<td>mebi-</td>
<td>2^20 (1,048,576)</td>
</tr>
<tr>
<td>giga-</td>
<td>10^9 (1,000,000,000)</td>
</tr>
<tr>
<td>gibi-</td>
<td>2^30 (1,073,741,824)</td>
</tr>
<tr>
<td>tera-</td>
<td>10^12 (1,000,000,000,000)</td>
</tr>
<tr>
<td>tebi-</td>
<td>2^40 (1,099,511,627,776)</td>
</tr>
<tr>
<td>peta-</td>
<td>10^15 (1,000,000,000,000,000)</td>
</tr>
<tr>
<td>pebi-</td>
<td>2^50 (1,125,899,906,842,624)</td>
</tr>
<tr>
<td>exa-</td>
<td>10^18 (1,000,000,000,000,000,000)</td>
</tr>
<tr>
<td>exbi-</td>
<td>2^60 (1,152,921,504,606,846,976)</td>
</tr>
<tr>
<td>zeta-</td>
<td>10^21 (1,000,000,000,000,000,000,000)</td>
</tr>
<tr>
<td>zebi-</td>
<td>2^70 (1,180,591,620,717,411,303,424)</td>
</tr>
</tbody>
</table>

corresponds to an integer value of no more than 2^{64}-1, or 1.84467440737e+19. (More on this later.) Older computers can only process 32 bits (an integer of 2^{32}-1, or 4,294,967,295). Still older computers and many modern micro-controllers are limited to 16 bits (an integer of 65,535).

It is possible for a CPU to process values larger than it’s word size. It simply can’t do it all at once (in a single operation). A 32-bit computer can process 64-bit numbers, but it has to do it 32-bits at a time, and hence it takes twice as long.

In fact, software exists for processing arbitrarily large (arbitrary precision) values. The bc calculator command, which is a standard part of every Unix system, is an example. Just be aware that processing arbitrarily large numbers takes an arbitrarily long time.

Another limit imposed by a CPU’s word size is how much memory it can address. The size of a memory address in most CPUs is the same as its word size. Hence, most 32-bit CPUs can address no more than 2^{32} bytes (4 gibibytes) of RAM. Many computers today use much more than this. A 64-bit CPU can theoretically address up to 2^{64} bytes (16 exbibytes), which is far more than today’s computers can have installed.

### 17.4 Fixed Point Number Systems

A fixed point number system has a fixed number of digits and a fixed number of fractional digits.

Such as system has a limited range and accuracy. As with any number set, the results of operations on two of the numbers must also be within the set. This can sometimes yield different results than we would see for an infinite number system.

**Example 17.1** A fixed-point system

Suppose a system has 4 decimal digits, two of which are fractional. Then all numbers have the form #.##.

The largest value we can represent is 99.99 and no value can have more than 4 significant figures.

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>94.01</td>
<td>10.00</td>
<td>03.33</td>
</tr>
<tr>
<td>+ 12.58</td>
<td>/ 03.00</td>
<td>* 03.00</td>
</tr>
<tr>
<td>--------</td>
<td>--------</td>
<td>--------</td>
</tr>
<tr>
<td>06.59</td>
<td>03.33</td>
<td>09.99</td>
</tr>
</tbody>
</table>

The answer 06.59 is not the same as we would get using real numbers. We have to drop the ’1’ carried over from the addition of the leftmost digits to fit within our fixed-point system. This is known as overflow error.
The answer 10.00 / 03.00 = 03.33 is not precise. Using real numbers, the result would be 3.33333... with the digits going on forever. On our fixed-point system, we have to stop after 2 fractional digits. This is called round-off error.

Note also that reversing the operation by multiplying the result by 03.00 does not produce the original value of 10.00. In fact, the inaccuracy in the result 0.33 (0.003333...) is multiplied by 3 along with the value itself, so the error in the final result of 0.99 is
0.1. This reveals a potential problem in computer programs that perform many calculations using limited number systems. The error can grow out of control!

17.5 Modular Number Systems

A modular integer system is a special case of fixed-point systems where there are no fractional digits. It operates within a subset of non-negative integers.

Modular systems are important because all computers use them to approximate integer operations. In order to represent true integers, which can have an infinite number of digits, a computer would require an infinite amount of memory to store those digits.

Example 17.2 A modulo-100 System

To illustrate the concept, consider a subset of the non-negative integers limited to 2 decimal digits. Such a set only includes values from 0 to 99, and is known as the modulo-100 set. The value 100 is known as the modulus.

<p>| 94 | 10 | 03 |</p>
<table>
<thead>
<tr>
<th>+ 12</th>
<th>/ 03</th>
<th>* 03</th>
</tr>
</thead>
<tbody>
<tr>
<td>06</td>
<td>03</td>
<td>09</td>
</tr>
</tbody>
</table>

Note that the same problems with overflow and round-off occur with any fixed-point system. The integer addition result of 106 is too large to fit in 2 digits, so the ‘1’ is dropped and the result becomes 06.

When performing operations such as addition or multiplication on modulo systems, we follow the exact same procedure as we would for integers, but simply discard any extra digits.

Looking at it another way, when counting in a modulo system, after reaching the largest allowable value, we "wrap around" back to zero. I.e., 99 + 1 = 0 in modulo 100 arithmetic.

Computers behave the same way when performing "integer" operations. We must be aware that what we call integers in computer programming are actually modulo number systems, limited to a certain number of binary digits. They can and often do overflow, and this does not normally trigger an error condition in the program.

Programmers must be aware of the range of the modulo integer sets used by any computer, and take measures to ensure that overflows do not cause incorrect results.

17.6 Binary Data Representation

Computers store all data as patterns of 0’s and 1’s. Information systems using 0’s and 1’s are collectively known as binary information systems.

Each 0 or 1 in a binary value is called a bit, which is short for binary digit. This can sometimes be misleading, since not all bits actually represent digits in a number. A bit could be part of any type of information, including numbers, letters of the alphabet, hieroglyphics, RGB color values, etc.

A collection of 8 bits is called a byte. A byte is a very common unit of storage for electronic memory. It is usually the smallest chunk of data that programs process, although many languages support processing individual bits as well. Processing data smaller than a byte is generally not as easy as processing whole bytes.

A collection of 4 bits is called a nybble.

A word is the maximum amount of data a CPU can process at once, and is usually 1, 2, 4, or 8 bytes (8 to 64 bits).

Numeric data is stored using several different binary number formats, all of which use a finite number of binary digits (bits), and therefore are subject to overflow and round-off.
17.7 The Arabic Numeral System

As you probably know, we use a weighted-digit *position notation* called the Arabic system. (As opposed to the Roman numeral system.)

This system is convenient for performing mathematical manipulations.

A number represented in the Arabic system is a weighted sum of products. Each digit is multiplied by a power of the *radix*, or *base*, depending on its position. The power on the radix of decreases from left to right, and the digit just left of the *radix point* (decimal point in decimal numbers) always has a power of 0.

\[ 458.12 = 4 \times 10^2 + 5 \times 10^1 + 8 \times 10^0 + 1 \times 10^{-1} + 2 \times 10^{-2} \]

We use a radix of 10 because we have 10 fingers, and in the early days of the Arabic system, most math was done by counting fingers.

17.8 Binary Fixed Point

A binary fixed point system is another example of an Arabic numeral system. The only distinction from decimal is the radix. Binary uses a radix of 2 instead of 10.

We specify the radix (base) using a subscript on the number. From now on, we do not assume a base of 10.

\[ 1001.101_2 = 1 \times 2^3 + 0 \times 2^2 + 0 \times 2^1 + 1 \times 2^0 + 1 \times 2^{-1} + 0 \times 2^{-2} + 1 \times 2^{-3} \]

\[ = 8 + 0 + 0 + 1 + .5 + 0 + .125 \]

\[ = 9.625_{10} \]

An easy way to read binary is by working left and right from the binary point, doubling or halving the value of the digits at each step:

\[
\begin{align*}
1001 & = 1 \times 1 + 0 \times 2 + 0 \times 4 + 1 \times 8 = 9 \\
.101 & = 1 \times .5 + 0 \times .25 + 1 \times .125 = 0.625
\end{align*}
\]

The digits used in any Arabic system always range from 0 to radix-1. Using larger digits would make it possible to represent a value in more than one way. For example, if binary fixed point allowed the use of the digit ‘2’, then the number two could be represented as either ‘2’ or ‘10’.

17.8.1 Limitations of Binary

Convert 1/10 to binary. Can’t be done: like trying to represent 1/3 in decimal. It requires an infinite number of digits. This is a problem for monetary calculations. Solutions? (BCD, cents)

17.9 Unsigned Binary Integers

An unsigned binary integer is a fixed-point system with no fractional digits.

Unsigned binary integers are modulo number systems, usually with a modulus which is a power of 2.

**Example 17.3** A 4-bit unsigned binary number system

A 4-bit unsigned binary number has values ranging from 0000₂ (0₁₀) to 1111₂ (15₁₀). Hence, it is a modulo-10000₂, or modulo-16₁₀.

Modern computers typically support binary integers of 8, 16, 32, or 64 bits.

The largest value in any unsigned binary integer system is the one containing all 1’s, just as the largest decimal number is the one containing all 9’s.
Largest modulo-1000 \(_{10} = 999_{10}.

Largest modulo-1000 \(_2 = 111_2.

If we add 1 to a binary value of all 1’s, note what happens:

\[
\begin{array}{c}
111 \\
1111 \\
+ 1 \\
\hline
10000
\end{array}
\]

If we add one to the largest value possible in 4 bits, we get 2\(^4\). If we add 1 to the largest value possible in \(N\) bits, we will get 2\(^N\). Hence, the largest value possible in \(N\) bits, is 2\(^N\)-1.

<table>
<thead>
<tr>
<th>Bits</th>
<th>Range</th>
</tr>
</thead>
<tbody>
<tr>
<td>8</td>
<td>0 to 2^8-1 (255)</td>
</tr>
<tr>
<td>16</td>
<td>0 to 2^16-1 (65,535)</td>
</tr>
<tr>
<td>32</td>
<td>0 to 2^32-1 (4,294,967,295)</td>
</tr>
<tr>
<td>64</td>
<td>0 to 2^64-1 (18,446,744,073,709,551,615 (= 1.844674407 \times 10^{19}))</td>
</tr>
</tbody>
</table>

Table 17.3: Unsigned Binary Integer Ranges

Note that even a 64-bit integer cannot hold Avogadro’s constant, 6 \(\times\) 10\(^{23}\). In order to represent very large numbers with only 32 or 64 bits, which must find a system other than fixed-point. This is discussed in Section 17.11.

### 17.10 Two’s Complement

#### 17.10.1 Format

The most common format used to represent signed integers in modern computers is *two’s complement*.

A positive integer in two’s complement always has a 0 in the leftmost bit (*sign bit*) and is represented the same way as an unsigned binary integer.

\(+14_{10} = 01110\) _two's comp_

#### 17.10.2 Negation

To negate a number, a process sometimes called “taking the two’s complement”, we invert all the bits and add one.

\(-14_{10} = 10001 + 1 = 10010\) _two's comp_

Note that the same process works for both positive and negative numbers. Subtracting one and inverting produces the same results as inverting and adding 1.

\[10010 - 1 = 10001, \text{ inverted } = 01110, 10010 \text{ inverted } = 01101, + 1 = 01110\]

| \(-0001\) | \(-1110 + 1 = 1111\) |
| \(-1111\) | \(-0000 + 1 = 0001\) |
| \(-1110\) | \(-0001 + 1 = 0010\) |
| \(-0010\) | \(-1101 + 1 = 1110\) |
| \(-1000\) | \(-0111 + 1 = 1000\) _Oops!_

Convert the following 4-bit 2’s comp values to decimal:
0111 = +(1 + 2 + 4) = +7
1000 = -(0111 + 1) = -(1000) = -8
0110 = +(2 + 4) = +6
1001 = -(0110 + 1) = -0111 = -(1 + 2 + 4) = -7
1110 = -(0001 + 1) = -0010 = -2

17.10.3 Addition and Subtraction

Addition works exactly like unsigned addition. This means a computer that uses 2’s complement to store signed integers can use the same adder circuit to do both signed and unsigned addition. Subtraction is done by negating and adding.

<table>
<thead>
<tr>
<th>Binary</th>
<th>Unsigned</th>
<th>2’s comp</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 1</td>
<td>0101</td>
<td>5</td>
</tr>
<tr>
<td>+ 1001</td>
<td>+ 9</td>
<td>+ -7</td>
</tr>
<tr>
<td>1110</td>
<td>14</td>
<td>-2</td>
</tr>
</tbody>
</table>

17.10.4 Range

Two’s complement essentially takes one bit away from the value for use as a sign bit. Since we have one fewer binary digit, the maximum value is 1/2 what it would be for an unsigned number with the same number of bits.

The largest positive value in N-bit two’s complement is 0111...111, which is $2^{N-1}-1$.

The smallest negative value in N-bit two’s complement is 1000...000, which is -$2^{N-1}$.

<table>
<thead>
<tr>
<th>Bits</th>
<th>Range</th>
</tr>
</thead>
<tbody>
<tr>
<td>8</td>
<td>-2^7 (-128) to +2^7-1 (+127)</td>
</tr>
<tr>
<td>16</td>
<td>-2^31 (-2,147,483,648) to +2^31-1 (+2,147,483,647)</td>
</tr>
<tr>
<td>32</td>
<td>-2^63 (-9,223,372,036,854,775,808) to +2^63-1 (9,223,372,036,854,775,807)</td>
</tr>
</tbody>
</table>

Table 17.4: Two’s Complement Integer Ranges

17.10.5 Comparison

Comparison of two’s complement values is not the same as unsigned comparison if the signs are different:

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>Unsigned</th>
<th>Two’s comp</th>
</tr>
</thead>
<tbody>
<tr>
<td>0111</td>
<td>0110</td>
<td>&gt;</td>
<td>&gt;</td>
</tr>
<tr>
<td>1111</td>
<td>1000</td>
<td>&gt;</td>
<td>&gt;</td>
</tr>
<tr>
<td>0111</td>
<td>1111</td>
<td>&lt;</td>
<td>&gt;</td>
</tr>
</tbody>
</table>

This is due to the fact that two’s complement rearranges the binary patterns on the number line, so that the latter half of the patterns (those beginning with 1, are less than the first half).

<table>
<thead>
<tr>
<th>Unsigned</th>
<th>Two’s comp</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>+7</td>
</tr>
<tr>
<td>0111</td>
<td>-8</td>
</tr>
<tr>
<td>1000</td>
<td>-1</td>
</tr>
<tr>
<td>1111</td>
<td>(out of sequence)</td>
</tr>
</tbody>
</table>
17.10.6 Overflow Detection

Overflow in two’s complement is determined by a result with the wrong sign. I.e., if you add two positives and get a negative result, or add two negatives and get a positive.

It is not possible to get an overflow when adding numbers of opposite signs!

<table>
<thead>
<tr>
<th></th>
<th>111</th>
<th>+7</th>
<th>111</th>
<th>+7</th>
<th>1111</th>
<th>-1</th>
</tr>
</thead>
<tbody>
<tr>
<td>+</td>
<td>0011</td>
<td>+3</td>
<td>1000</td>
<td>-8</td>
<td>1000</td>
<td>-8</td>
</tr>
<tr>
<td></td>
<td>1010</td>
<td>-6</td>
<td>1111</td>
<td>-1</td>
<td>0111</td>
<td>+7</td>
</tr>
</tbody>
</table>

17.10.7 Extension and Reduction

Two’s complement values are extended to larger formats by copying the sign bit to all new positions:

<table>
<thead>
<tr>
<th>4-bit</th>
<th>8-bit</th>
<th>16-bit</th>
<th>Decimal</th>
</tr>
</thead>
<tbody>
<tr>
<td>0111</td>
<td>00000111</td>
<td>00000000000000111</td>
<td>+7</td>
</tr>
<tr>
<td>1110</td>
<td>11111110</td>
<td>111111111111111110</td>
<td>-2</td>
</tr>
</tbody>
</table>

17.11 Floating Point

17.11.1 The Basics

Floating point gets around the limitations of fixed point by using a format similar to scientific notation.

3.52 x 10^3 = 3520

A scientific notation number, as you probably know, consists of a mantissa (3.52 in the example above) a radix (always 10), and an exponent (3 in the example above). Hence, the general format of a scientific notation value is:

mantissa x radix^exponent

The normalized form always has a mantissa greater than or equal to 1.0, and less than 10.0. We can denormalize the value and express it in many other ways, such as 35.2 x 10^2, or 0.0325 x 10^5. For each position we shift the digits of the mantissa relative to the decimal point, we increase or decrease the value of the mantissa by a factor of 10. To compensate for this, we simply increase or decrease the exponent by 1. Denormalizing is necessary when adding scientific notation values:

```
3.52 x 10^3
+ 1.97 x 10^5
-----------------------------------------
1
0.0352 x 10^5
+ 1.97 x 10^5
-----------------------------------------
2.0052 x 10^5
```

Adjusting the mantissa and exponent is also sometimes necessary to normalize results. For example, 9.9 x 10^2 + 9.9 x 10^2 is 19.8 x 10^2, which must be normalized to 1.98 x 10^3.

A binary floating system stores a signed binary mantissa and a signed binary exponent, and usually uses a radix of 2. Using a radix of 2 (or any power of 2) allows us to normalize and denormalize by shifting the binary digits in the mantissa and adjusting the integer exponent on the radix of 2. (Shifting binary digits in the mantissa n bits to the left or right multiplies or divides the mantissa by 2^n.)

00010₂ x 2³ = 01000₂ x 2¹.

The standard floating point formats are defined by the IEEE society. The IEEE formats are slightly more complex that necessary to understand floating point in general, so we will start with a simpler example here.
17.11.2 A Simple Floating Point Format

Suppose a 32-bit floating point format has a 24-bit two’s complement mantissa, an 8-bit two’s complement exponent, and a radix of 2. The general structure is:

mantissa \times 2^{exponent}

Where mantissa is a 24-bit two’s complement integer, and exponent is an 8-bit two’s complement integer.

The binary format is as follows:

<table>
<thead>
<tr>
<th>Mantissa</th>
<th>Exponent</th>
</tr>
</thead>
<tbody>
<tr>
<td>24 bits, two’s compliment</td>
<td>8 bits, two’s compliment</td>
</tr>
</tbody>
</table>

Table 17.5: Floating Point Format

1. What is the value of the following number?  
   
   \begin{align*}
   000000000000000000010010 11111100
   \end{align*}
   
   The mantissa is 000000000000000000010010, or +(2 + 16) = +18.  
   The exponent is 11111100 = -(00000011 + 1) = -00000100 = -4.  
   The value is therefore +18 \times 2^{-4}

2. What is the largest positive value we can represent in this system?  
   
   The largest positive value will consist of the largest positive mantissa and the largest positive exponent.  
   The largest mantissa is 011111111111111111111111, which in two’s complement is +2^{23} - 1 (+8388607).  
   The largest exponent is 01111111, which in two’s complement is +2^7 - 1 (+127).  
   Hence, the largest positive value is +8388607 \times 2^{+127} = 1.42 \times 10^{45}.

3. What is the second largest positive value? What is the difference between the largest and second largest?

4. What is the smallest positive value?  
   
   To find the smallest positive value in the form mantissa \times \text{radix}^{exponent}, we choose the smallest positive mantissa, and the smallest negative exponent (the negative exponent with the greatest magnitude).  
   Since the mantissa is an integer, the smallest positive value possible is 1.  
   Since the exponent is an 8-bit two’s complement value, the smallest negative exponent is 10000000_2, of -2^7 = -128.  
   Hence the smallest positive value is 1 \times 2^{-128}, or 2.93873587706 \times 10^{-39}.

5. What is the second smallest positive value? What is the difference between the smallest and second smallest?

6. Represent -2.75 in this floating point system.
   
   (a) Convert the number to fixed point binary using the methods described in previous sections:  
   \begin{align*}
   -2.75_{10} &= -(10.112) \\
   \text{(b) Multiply by } \text{radix}^{exponent} \text{ equal to 1:} \quad -2.75_{10} &= -(10.112) \times 2^0 \\
   \text{(c) Shift the binary point to make the mantissa a whole number: } -(10112) \\
   \text{(d) Convert the mantissa and exponent into the specified formats (two’s complement in this case):} \\
   \text{Mantissa: } -(00000000000000000010111111111111101101) = 11111111111111110101111111110 \\
   \text{Exponent: } -2_{10} = 11111111 \\
   \text{Binary representation } = 1111111111111110101111111111110
   \end{align*}

7. How many different values can this system represent?
17.11.3 Overflow and Underflow

*Overflow* occurs when the result of a floating point operation is larger than the largest positive value, or smaller than the smallest negative value. In other words, the magnitude is too large to represent.

*Underflow* occurs when the result of a floating point operation is smaller than the smallest positive value, or larger than the largest negative value. In other words, the magnitude is too small to represent.

The example 32-bit format above cannot represent values larger than about $10^{45}$ or smaller than about $10^{-39}$.

One technique to avoid overflow and underflow is to alternate operations that increase and decrease intermediate results. Rather than do all the multiplications first, which could cause overflow, or all the divisions first, which could cause underflow, we could alternate multiplications and divisions to moderate the results along the way. Techniques like these must often be used in complex scientific calculations.

17.11.4 Cost of Floating Point

Everything has a cost. The increased range and ability to represent non-whole numbers is no exception.

**Precision**

There are only $2^{32}$ patterns of 32 0’s and 1’s. Hence, there are only $2^{32}$ unique numbers that we can represent in 32 bits, regardless of the format.

So how is it we can represent numbers up to $10^{45}$?

Obviously, we must be sacrificing something in between. What floating point does for us is spread out the limited number of binary patterns we have available to cover a larger range of numbers. The larger the exponent, the larger the gap between consecutive numbers that we can accurately represent.

The gap between consecutive numbers that we can represent gets larger as we move away from 0.

The precision of a 32-bit floating point value is less than the precision of a 32-bit integer. By using 8 bits for the exponent, we sacrifice those 8 bits of precision. Hence, our example format has the same precision as a 24-bit signed integer system.

**Performance**

Arithmetic on floating point is several times slower than on integers. This is an inherent property of the format.

Consider the process of adding two scientific notation values.

1. Equalize the exponents
2. Add the mantissas
3. Normalize the result

Each of these operations take roughly the same amount of time in a computer as a single integer addition. Since floating point is stored like scientific notation, we can expect floating point addition to take about three times as long as integer addition. In reality, a typical PC takes about 2.5 times as long to execute a floating point arithmetic instruction as it does to do the same integer instruction.

Note that this applies only to operations that can be carried out using either a single integer instruction or a single floating point instruction. For example, suppose a program is running on a 32-bit computer, and there is no way to represent the data within the range of a 32-bit integer. In this case, multiple integer instructions will be necessary to process integer values of more than 32 bits, and the speed advantage of integers does not apply.

It is also possible in some systems that floating point and integer operations could occur at the same time, and hence utilizing the floating point hardware could result in better performance than performing additional integer operations while the floating point unit sits idle. This is the case with graphics rendering that occurs using floating point on the graphics processing unit (GPU).
rather than the CPU. It would not make sense to move the rendering calculations to the CPU in order to use integers, as this would only increase the workload for the CPU and allow the power of the GPU to go to waste.

If hardware has floating point support built-in, then common operations like floating point addition, subtraction, etc. can each be handled by a single instruction. If hardware doesn’t have a floating point unit (common in embedded processors), floating point operations must be handled by software routines. Hence, adding two floating point values will require dozens of instructions to complete instead of just one. These will be hundreds of times slower than integers, and will consume a big chunk of available program memory.

Most algorithms can be implemented using integers with a little thought. Use of floating point is often the result of sheer laziness. Don’t use floating point just because it’s intuitive.

More power consumption. CPUs achieve their maximum power consumption when doing intensive floating point calculations. This is not usually noticeable on a desktop PC, but can become a problem on large grids consisting of hundreds of PCs, since the power grid they are attached to may not be designed to provide for their maximum draw. It can also be a problem when running a laptop on battery while doing intensive computations. Battery life while doing intensive floating point computations could be a small fraction of what it is while reading email, browsing the web, or editing a document in OpenOffice.

17.12 Character Storage

17.12.1 ASCII

Characters are stored as a standardized set of binary patterns based on ASCII (American Standard Code for Information Interchange). ASCII is a 7-bit binary code. Although ASCII values are not numbers, they are often treated as such, and represented in decimal for readability.

<table>
<thead>
<tr>
<th>0-31</th>
<th>Control chars. Originally designed to control text-only printers. Adopted by ASCII terminals (vt100, xterm) for analogous functions.</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>EOT (ctrl+d)</td>
</tr>
<tr>
<td>7</td>
<td>BEL (ctrl+g) \007 Beep printer/terminal</td>
</tr>
<tr>
<td>8</td>
<td>BS (ctrl+h) \b Move head/cursor left</td>
</tr>
<tr>
<td></td>
<td>May shift rest of line on terminal</td>
</tr>
<tr>
<td>9</td>
<td>TAB (ctrl+i) \t Move head/cursor to next tab stop</td>
</tr>
<tr>
<td>10</td>
<td>LF (ctrl+j) \n Move head/cursor down (scroll)</td>
</tr>
<tr>
<td>12</td>
<td>FF (ctrl+l) \f Scroll to start of next page</td>
</tr>
<tr>
<td>13</td>
<td>CR (ctrl+m) \r Move head/cursor to col 1</td>
</tr>
</tbody>
</table>

Terminals: CR+LF goes to start of new line. Unix adds CR to each LF by default, so we write "Hello, world!" instead of "Hello, world!
".

<table>
<thead>
<tr>
<th>32-126</th>
<th>Printable characters. Print/display char and move head/cursor right. Wrap at EOL of printer/terminal supports it.</th>
</tr>
</thead>
<tbody>
<tr>
<td>32</td>
<td>Space</td>
</tr>
<tr>
<td>33</td>
<td>!</td>
</tr>
<tr>
<td>...</td>
<td></td>
</tr>
<tr>
<td>48</td>
<td>’0’</td>
</tr>
<tr>
<td>49</td>
<td>’1’</td>
</tr>
<tr>
<td>...</td>
<td></td>
</tr>
<tr>
<td>65</td>
<td>’A’</td>
</tr>
<tr>
<td>66</td>
<td>’B’</td>
</tr>
<tr>
<td>...</td>
<td></td>
</tr>
<tr>
<td>97</td>
<td>’a’</td>
</tr>
<tr>
<td>98</td>
<td>’b’</td>
</tr>
<tr>
<td>127</td>
<td>DEL Delete char under cursor</td>
</tr>
</tbody>
</table>
Examples:
char ch = 'A';
ch: .byte 'A'

What is in the variable ch? (binary)

<table>
<thead>
<tr>
<th>Code</th>
<th>Dec Char</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000000</td>
<td>0 NUL</td>
</tr>
<tr>
<td>0000100</td>
<td>4 EOT</td>
</tr>
<tr>
<td>0000111</td>
<td>7 BEL</td>
</tr>
<tr>
<td>0001000</td>
<td>8 BS</td>
</tr>
<tr>
<td>0001001</td>
<td>9 TAB</td>
</tr>
<tr>
<td>0001010</td>
<td>10 NL</td>
</tr>
<tr>
<td>0001101</td>
<td>13 CR</td>
</tr>
<tr>
<td>0100000</td>
<td>32 Space</td>
</tr>
<tr>
<td>0100001</td>
<td>33 !</td>
</tr>
<tr>
<td>0110000</td>
<td>48 0</td>
</tr>
<tr>
<td>0110001</td>
<td>49 1</td>
</tr>
<tr>
<td>...</td>
<td></td>
</tr>
<tr>
<td>1000001</td>
<td>65 A</td>
</tr>
<tr>
<td>1000010</td>
<td>66 B</td>
</tr>
<tr>
<td>...</td>
<td></td>
</tr>
<tr>
<td>1100001</td>
<td>97 a</td>
</tr>
<tr>
<td>1100010</td>
<td>98 b</td>
</tr>
<tr>
<td>...</td>
<td></td>
</tr>
</tbody>
</table>

17.12.2 ISO

The International Standards Organization (ISO) extended the ASCII set to 8 bits to include non-English characters, and eventually non-Latin-based characters such as found in Asian languages. ISO-Latin1 is the most common character set used in the west. The UTF, or Unicode character sets use multiple bytes to represent more than 256 characters, which is necessary for languages such as Mandarin.

17.12.3 Unicode

Unicode is a computing industry standard for representing the characters in most of the world’s writing systems. It currently consists of over 100,000 characters from more than 90 writing scripts.

The Unicode Transformation Formats (UTF) provide ways to encode the Unicode character set using a stream of bytes. UTF-8 is the most commonly used format, and is backward-compatible with ASCII. UTF-8 encodes each of the Unicode characters using one to four bytes.

17.13 Self-test

4. Why do we study data representation?

5. List the binary orders of magnitude.

6. Perform the following operations in modulo-1000 integer arithmetic. Indicate whether an overflow occurs in each case, and why.
   - 45 * 12
7. Perform the following operations in modulo-1000 integer arithmetic. Indicate whether an overflow occurs in each case, and why.
   - $794 + 318$

8. Using a fixed point system with four decimal digits, and three whole digits, compute each of the following. Indicate in each case whether the result is the same as when computed with real numbers, and if not, why.
   - $5 / 2 * 4$
   - $1 / 4 * 10$

9. Define each of the following:
   - bit
   - byte
   - nybble
   - word

10. What type of number system is the Arabic system?

11. Why do we use the Arabic system instead of the Roman numeral system?

12. What is the value of $10001010.0112$?

13. What limitations does binary have when it comes to representing real numbers? Provide one example, and a comparable example of the limitations of decimal.

14. What is the modulus of a 10-bit unsigned binary number system?

15. What is the decimal value of $100100102$?

16. What is the range in decimal of a 12-bit unsigned binary integer?

17. Perform the following 8-bit unsigned addition. Indicate whether there is an overflow, and why.

   \[
   \begin{array}{c}
   10010001 \\
   + \quad 01110010 \\
   \hline
   \end{array}
   \]

   \[= 11110010\]

18. What are the decimal values of the following 4-bit two’s complement numbers?
   - 1001
   - 0110
   - 1000

19. Represent the following values in 8-bit two’s complement:
   - +45
   - -57
   - -135

20. What is the range of a 9-bit two’s complement integer?
21. Perform the following 4-bit two’s complement additions. Indicate in each case whether there is an overflow, and why.

<table>
<thead>
<tr>
<th>1001</th>
<th>1110</th>
<th>0011</th>
</tr>
</thead>
<tbody>
<tr>
<td>+</td>
<td>0110</td>
<td>+</td>
</tr>
</tbody>
</table>

22. Extend the following 4-bit two’s complement values to 8 bits.
   - 0111
   - 1101

23. A floating point system consists of a 7-bit two’s complement exponent (rightmost 7 bits), and a 25-bit two’s complement mantissa. The radix is 4.
   - What is the largest positive value the system can represent?
   - What is the smallest positive value the system can represent?
   - What is the decimal value of 11111111111111111111101001111010
   - Represent +0.75 in this floating point format.

24. What does ASCII stand for?

25. Describe the major subsets of characters within the ASCII set, and provide the starting and ending values in decimal, two examples of each (name and their decimal value).

26. What is ISO-Latin1?

27. What is Unicode?

28. What is UTF?
Chapter 18

Computer Hardware

18.1 Motivation

A basic understanding of computer hardware is essential to understand what computer programs do, and how to make them efficient.

18.2 What is a Computer?

A computer is a general purpose computing machine. It can carry out complex sequences of calculations, as instructed by a program.

- CPU: Central Processing Unit
- RAM: Random Access Memory (Read/Write Memory), volatile
- ROM: Read-only Memory, non-volatile
- Mass/Auxiliary storage: Disk, tape, Flash, non-volatile
- Input/output devices: Human Interface Devices (HIDs), computer-computer communication (network), computer-device communication (USB, Firewire, Bluetooth)

18.3 Central Processing Unit

The Central Processing Unit, or CPU directs the activities of all other hardware. The CPU is in turn directed by programs, which are sequences of instructions.

18.4 Non-volatile Auxiliary Storage: Disk, Tape and Flash

Non-volatile storage is storage that retains its contents while the power is off. This includes magnetic disks (hard disks), optical discs such as CD, DVD, and Blu-Ray, Flash devices such as USB thumb drives, and magnetic tape.
18.5 Electronic Memory

18.5.1 RAM and ROM

Electronic memory is categorized into two major types:

- RAM (Random Access Memory), is both readable and writable. It is volatile, which means it requires power in order to retain its contents. (The term volatile is borrowed from chemistry, where it refers to something that evaporates.) Computer programs and data are normally stored in disk files when they are not in use. When a program is run (executed), the program and the data it manipulates are loaded from disk into RAM, which is about 1,000,000 times faster than disk. This greatly improves speed when the same data is accessed repeatedly.

- ROM (Read-Only Memory) is not generally writable. The original ROMs were set to a specific content at the factory and could never be written again.

Today, we use EEPROM, or electronically erasable programmable ROMs, such as Flash memory. EEPROMS are writable, but not as easily as RAM. There are special procedures to altering the contents of a EEPROM. The important feature of ROM and EEPROM is that it is non-volatile, so it retains its content even when the power is cut.

Non-volatile memory is used to store firmware, which is essentially software that stays in memory when the power is off. Firmware makes it possible for computers to start when the power is turned on (cold boot), and allows small and embedded devices which are often powered down to function. The boot sequence cannot be started from disk, since reading a program from disk requires a program! Hence, there must be a minimal amount of program code in memory when the power comes on to start the boot sequence. In a personal computer, the firmware, sometimes called BIOS (Basic Input/Output System or Built-In Operating System), initializes the hardware and loads the first part of the operating system from disk. From there, the operating system takes over.

18.5.2 Memory Structure

Memory is a one-dimensional array of storage cells. In virtually all computers today, each cell holds one byte. Memory cells are selected using an integer index called the memory address. Memory addresses begin at zero.

<table>
<thead>
<tr>
<th>Address</th>
<th>Content</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>01001010 (decimal 74, character 'c', or anything else!)</td>
</tr>
<tr>
<td>1</td>
<td>01010010</td>
</tr>
<tr>
<td>2</td>
<td>11010101</td>
</tr>
<tr>
<td>3</td>
<td>00001011</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>Memory size - 1</td>
<td>10100110</td>
</tr>
</tbody>
</table>

Table 18.1: Example Memory Map

Each 8-bit cell can hold a character such as 'a' or '?', an integer between 0 and 255, or part of a larger piece of information such as a 64-bit floating point number, which would occupy eight consecutive cells.

18.5.3 Virtual Memory

Virtual memory uses disk to extend the amount of RAM that is apparently available to programs.

When a computer using virtual memory runs out of RAM, it swaps blocks of data from RAM out to a special area of disk known as swap space.

Since disk is typically 1,000 to 1,000,000 times slower than RAM, swapping is very expensive. A single swap to or from disk takes only a small fraction of a second, but this is far longer than RAM access takes, so if a program does many swaps, it can become unbearably slow. Swap space is therefore useful mainly for inactive programs such as word processors, which spend most of their time waiting for user input. Parts of the program or data that are not actively in use can be swapped out to disk to
make room for more active programs. If it takes a fraction of a second to swap something back into RAM when the user presses a key or clicks an icon, the user won’t generally notice.

For very active programs, swap is of little use. Programs that actively use more memory than the system has available as RAM may spend most of their time waiting for swap operations. When this happens, the system is said to be **thrashing**, like someone struggling to stay afloat, but not moving forward through the water.

Therefore, it is important for computationally intensive programs to use as little memory as possible.

### 18.6 Computer Word Size

What is a “64-bit” computer?

The word size of a computer generally indicates the largest integer it can process in a single instruction, and the size of a memory address, which is usually, but not necessarily the same as the integer size.

The main indication of the word size is how much memory the processor can address. The maximum amount of memory a processor is capable of accessing is known at the **address space**.

A processor with a 32-bit address, such as an Intel Pentium or AMD Athlon, can only represent $2^{32}$ distinct memory addresses. Hence, if each address contains one byte, then the computer can have at most $2^{32}$ bytes, or 4 gibibytes (a little more than 4 billion bytes) of memory. This is more than enough RAM for most computer users, but some applications can benefit from using much more, so 64-bit processors and cheap RAM are blessings to come types of computing.

Note that 32-bit processors such as the 486, Pentium, and PowerPC G4 have supported 64-bit floating point numbers for a long time, but were still regarded as 32-bit processors due to their 32-bit integers and addresses.

An AMD64 processor has a 64-bit memory address, and uses byte-addressable memory (one byte of memory per address). Hence, the memory address space is $2^{64}$ bytes (16 exbibytes, or roughly 16 billion gibibytes). Recall that a gibibyte is $2^{30}$ bytes, or a little more than a gigabyte ($10^9$).

### 18.7 Self-test

1. What does the CPU do?
2. What is RAM used for and why?
3. What is non-volatile memory used for and why?
4. What limits the size of a computer’s address space? How is this related to the actual amount of memory in a computer?
5. What does virtual memory do?
6. What is virtual memory good for and not good for? Why?
Chapter 19

Introduction to Programming

19.1 C and C++

C and its offshoot, C++, are the most popular compiled programming languages of recent decades.

C is a very simple, but high-level language that is renowned for giving the programmer unlimited power and flexibility, near optimal execution speed, and maximizing code density, which means you can attain a lot of functionality with a minimal amount of typing.

Code written in C can be easily utilized by code written in other languages such as C++ and Fortran, which makes C a good language for writing general-purpose code, especially library functions.

C++ is a superset of C, which means that C source code can be used directly in C++ programs. Unlike C, the C++ language has explicit support for object-oriented programming, plus other enhancements that benefit the development of complex applications.

Note Object-oriented programming is a design discipline, not a language selection. It is possible to write object-oriented programs and any language. Likewise, it is possible to write non-object-oriented code in an object-oriented language. Simply using an object-oriented language does not make you an object-oriented programmer.

The object-oriented features of C++ are most beneficial to programmers dealing with complex data structures, which are more common in business than in scientific programming. C++ programs are also noticeably larger and marginally slower than the same program written in C. This should be weighed against the benefits of using C++ when choosing a language for your applications. Nevertheless, many powerful scientific applications and libraries are being written in C++.

This introduction will focus on C in order to avoid confusing readers who are not experts in programming with the complexities of C++. Since C is a subset of C++, everything discussed here will directly benefit those who want to continue on and learn C++.

19.1.1 Self-test

1. How widely used are C and C++?
2. How fast can C and C++ programs be? Explain.
4. What is C known for?
5. What is the relationship between C++ and C?
19.2 Not Your Grandfather's Fortran

Fortran was the first widely available high level language, originally created in the 1950's. If you tell people, especially computer scientists, that you're learning Fortran, you may get some odd looks and snide remarks such as "Who still uses Fortran?", or "My grandpa use to use that.".

What you're learning here is not your grandfather's Fortran, however. Fortran has gone through a number of major evolutionary steps, and is greatly enhanced since the early days. Fortran 90 brought some particularly important improvements, such as free-format (Fortran versions up to 77 requires a strict line structure) and more support for structured code. Fortran has always had intrinsic support for complex numbers, and newer versions support many matrix operations like those found in MATLAB and other higher-level tools.

Fortran is a compiled language, so well-written Fortran programs run about as fast as any program could.

Fortran is an open standard language, so there are many compilers available from multiple vendors. There are also free Fortran compilers for most common computer hardware and operating systems.

19.2.1 Self-test

1. Is Fortran a modern language? Explain.
2. How fast do Fortran programs execute? Why?

19.3 High-level Program Structure

19.3.1 C

The general layout of a simple C program includes the following components:

1. Block comment describing the program
2. One or more #include directives to include header files
3. Main program body (required)
   (a) int main(int argc, char *argv[])
   (b) {
   (c) Variable definitions/declarations + comments
   (d) Program statements + comments
   (e) A return statement
   (f) }

C and C++ are case-sensitive, so printf is not a valid substitute for printf.
Example 19.1 A Simple C Program

/***************************************************************************
 * Description:
 * Compute the area of a circle given the radius as input.
 * 
 * History:
 * Date    Name       Modification
 * 2013-07-28 Jason Bacon Begin
***************************************************************************/

#include <stdio.h>
#include <math.h> // Defines M_PI
#include <sysexits.h>

int main(int argc,char *argv[]) {
    // Variable definitions for main program
    double radius, area;
    // Main program statements
    printf("What is the radius of the circle? ");
    scanf("%lf", &radius);
    if ( radius >= 0 ) {
        area = M_PI * radius * radius;
        printf("The area is %.2f.\n", area);
    } else
        fprintf(stderr,"The radius cannot be negative.\n");
    return EX_OK;
}

C is a free format language, which means the compiler treats the end of a line the same as a space or tab. The end of a variable definition or statement is indicated by a semicolon (;).

19.3.2 Fortran

The general layout of a simple Fortran program includes the following components:

1. Block comment describing the program
2. Main program body (required)
   (a) program <name>
   (b) Optional use statements to add extended features
   (c) Variable definitions/declarations + comments
   (d) Program statements + comments
   (e) end program <name>

Fortran is not case sensitive, so end is the same as END or End.
Example 19.2 A Simple Fortran Program

```fortran
!=======================================================================
! Description:
! Compute the area of a circle given the radius as input.
!
! Modification history:
! Date Name Modification
! 2011-02-16 Jason Bacon Begin
!=======================================================================
!
! Main program body

program Circle_area
  use iso_fortran_env ! Enable error_unit for error messages

  ! Disable implicit declarations (i-n rule)
  implicit none

  ! Constants
  real(8), parameter :: PI = 3.1415926535897932d0

  ! Variable definitions for main program
  real(8) :: radius, &
            area

  ! Main program statements
  print *, 'What is the radius of the circle?'
  read *, radius
  if ( radius >= 0 ) then
    area = PI * radius * radius
    print *, 'The area is ', area
  else
    write(error_unit,*) 'The radius cannot be negative.'
  endif
end program
```

Fortran was originally designed as a line-oriented language, which means that the end of a line marks the end of a statement. Most newer languages, in contrast, are free-format, so that a single statement can span many lines, or multiple statements may be on the same line. Languages such as C, C++, and Java use a semicolon to mark the end of each statement, and line structure is completely ignored.

Fortran 90 introduced a more flexible source code format than previous versions, but still uses the end of a line to mark the end of a statement. If a particular statement is too long to fit on the screen, it can be continued on the next line by placing an ampersand at the end of the line to be continued:

```fortran
print *, 'This message is too long to fit on a single line, ', &
        'so we use the continuation character to break it up.'
```

19.3.3 Self-test

1. Describe the high-level structure of a Fortran program.
2. Describe the high-level structure of a C program.
3. What is a line-oriented language?
4. Can you write a statement in a Fortran program that spans multiple lines? If so, how?
5. What is a free-format language? How are statements separated in such a language?

19.4 The Compilation Process

As compiled languages, C, C++, and Fortran programs are translated entirely to machine language before being executed.

19.4.1 Compilation Stages

In all three languages, production of an executable file involves up to three steps:

1. Preprocessing: This step runs the source code through a stream editor designed specifically for editing source code called the preprocessor. The preprocessor makes modifications such as replacing named constants with their values and inserting source code from other files (called header files).
   
   The preprocessor command is usually `cpp` (short for C PreProcessor).
   
   The preprocessor is described in detail in Section 19.7.

2. Compilation: This step translates the preprocessed source code to machine language (also known as object code), storing the resulting machine code in an object file. The object file is not an executable file, as certain components necessary to load and run a program have not been added yet. Object files on Unix systems have a file name extension of ".o".

3. Linking: This step combines the machine code necessary to load and run a program with one or more object files to create a complete executable.
   
   The linker program is usually called `ld`.
   
   Object files may be ".o" files that are part of the program being built, or they may be retrieved from object code archives known as libraries. An example of a library is `/usr/lib/libc.so`, the standard C library. It contains the object files for many standard functions used in the C language.

You do not generally need to run these steps individually. They are executed automatically in sequence when you run a compiler such as `cc`, `gcc`, or `gfortran`.

```
source code --> cpp --> compiler ----+
     |                                   
     v                                   
source code --> cpp --> compiler --> ld --> executable
     |                                   
libraries -------------------------------+
```

19.4.2 Portability

Every Unix system with a C compiler has a `cc` command. On FreeBSD and OS X, `cc` is equivalent to `clang`. On Linux systems, `cc` is equivalent to `gcc`. On many commercial Unix systems, `cc` is a proprietary compiler developed by the company.

Unfortunately, there is no standard compiler name for Fortran, since most Unix system don’t include a Fortran compiler. Fortran is usually added in the form of `f2c` (a Fortran 77 to C translator), `g95` (an open source Fortran 95 compiler), `gfortran` (the open source GNU Fortran compiler), or `flang` (the open source Clang/LLVM Fortran compiler). There are also many commercial Fortran compilers available. Each of these are only supported on specific platforms, though, such as Windows, OS X, Redhat Enterprise Linux, etc.

C source files have an extension of ".c". C++ files usually use ".cc", ".cpp", ".cxx", or ".c++".

Fortran files use ".f", ".F", ".for", or ".FOR" for Fortran 77, ".f90" or ".F90" for Fortran 90, ".f03" or ".F03" for Fortran 2003, and ".f08" or ".F08" for Fortran 2008.
The commands above will produce an executable file called `a.out`. This is the default for most Unix compilers. The executable file is also sometimes called a binary file. This is why program directories on Unix systems are named "bin" (/bin, /usr/bin, /usr/local/bin, etc.)

The machine code in a binary file output by a compiler is also sometimes called object code.

To run the binary program, we simply type its file name followed by any arguments that it requires.

```
shell-prompt: ./a.out
```

Most Unix compilers also support the `-o` flag to specify a different output file name.

```
shell-prompt: cc jumping-genes.c -o jumping-genes
shell-prompt: ./jumping-genes
```

Clang/LLVM and GCC are both open source compiler suites and are highly compatible with each other. They support most of the same command-line flags, such as `-Wall` to enable all possible warning messages.

```
shell-prompt: cc -O2 jumping-genes.c -o jumping-genes
```

With some care, the same compile commands can be used to build a program on any Unix system. You may want to use some non-portable flags such as `-Wall` to help find bugs during development, but this flag is not portable, i.e. it is not recognized by all compilers. The instructions provided to others for compiling your program should usually use `cc`, not `clang` or `gcc`, and should be limited to portable flags such as `-O` and not include flags only supported by certain compilers, such as `-Wall`.

Using `-O` will usually improve the speed of your executable file significantly and will often reduce its size as well. The `-O2` will usually offer only a marginal improvement over `-O`, and `-O3` will usually provide little or no benefit over `-O2`. Higher optimization levels like 2 and 3 may also impede debugging, since they may reorganize the machine code in ways that make it impossible to determine which line of source code a given machine instruction came from.

Generally, the higher the level of optimization, the more dangerous and less beneficial the optimizations will be. Using `-O` will include all optimizations considered to be very safe and will provide the vast majority of all the performance benefit that’s possible.

You can also enable specific optimizations using other command line flags, but such flags will be specific to a given compiler and not portable.

Before committing to use anything more sophisticated than `-O2`, compare the run time of your program when compiled with various options to see if it’s really worth doing.

### 19.4.3 Using Libraries

Libraries, as mentioned above, are collections of precompiled subprograms that we can use in our programs. Libraries are built with the same compilers as executables (`cc`, `c++`, `gfortran`, `flang`, etc).

We can create our own libraries as described in Chapter 24. More often, we will use libraries supplied with the compiler or installed as packages using tools such as Debian’s `apt-get`, FreeBSD’s `pkg`, or the `port` command in MacPorts.

While all languages use libraries, the C language was intentionally designed to rely heavily on them. The C language designers decided not to give the language any features that could be implemented as a library function. This keeps the language very simple, fast, easy to learn, and easy to implement on new hardware. In some cases it makes the program a bit less elegant, but no harder to read in reality.

For example, to compare two strings in many languages, we might use something like this:
if ( string1 == string2 )

The C language does not provide string comparison, so for this we use a library function call:

if ( strcmp(string1, string2) == 0 )

There are two kinds of libraries, static and dynamic (also known as shared). On most Unix systems, static libraries have file names ending with .a and shared libraries have file names ending in .so. For example, the standard math library, which contains functions such as sine and cosine written in C, is available as /usr/lib/libm.a and /usr/lib/libm.so.

When we link our program with a static library, the machine code for functions from the library is copied into our executable file. When we link with a shared library, only a reference to the library is placed in our executable. A compatible shared library must therefore be present on every computer that runs the program. When the program is started, the system will look for shared libraries that it references in order to find all of the machine code required by the program.

Shared libraries save disk space, since they eliminate the need for redundant copies of machine code in every executable. It can also save memory at run-time, since multiple processes running completely different programs can share the same copy of a library function in memory. This is very useful for graphical user interfaces, where many programs share a hefty set of functions for displaying drop-down menus, title bars, etc.

The downside of shared libraries is that we have to make sure we install compatible shared libraries with each program that uses them. Using a package manager eliminates most of these issues. Programs using shared libraries may also take longer to start up, since the loader has to load the object code from multiple files, which is less efficient than reading a single file.

Some libraries, such as the standard C library (usually /usr/lib/libc.a or /usr/lib/libc.so) are automatically searched by the linker. For other libraries stored in default locations, such as the standard math library (usually /usr/lib/libm.a or /usr/lib/libm.so) we need only tell the linker to search it, by using the -l flag. This flag is immediately followed by the unique portion of the library's file name. For example, to use libm.a or libm.so, we specify -lm:

cc -O gauss.c -o gauss -lm

All library file names begin with "lib" and end with common extensions like ".a", ".so", or ".dylib". We omit these parts when using the -l flag.

Add-on libraries, such as those installed by a package manager, may not be in the linker's default search path, so we need to use -L to tell the linker where to find the library file. This flag is immediately followed by the absolute or relative path of the directory containing the library. For example, to use /usr/local/lib/libblas.a, we would use a compile command like the following:

cc -O gauss.c -o gauss -L/usr/local/lib -lblas -lm
gfortran -O gauss.f90 -o gauss -L/usr/local/lib -lblas -lm

Note Order may be important with -l flags. For example, if a function in the blas library calls a function in the standard math library, then -lm should come after -lblas.

19.4.4 Self-test

1. What does the preprocessor do?
2. What does the compiler do?
3. What is an object file?
4. Where does the linker get all the object files it needs to build an executable file?
5. What is an object code library?
6. What is the standard command for compiling C programs on Unix systems? What are some other, nonportable compiler commands?

7. What is the standard Unix command for compiling Fortran programs?

8. What are some examples of portable compiler flags with Unix compilers. List some examples and state what they do.

9. What is the default name of an executable file produced by Unix compilers? How can you override it?

10. What does the \texttt{-Wall} do? Is it supported by all compilers?

11. Write a C program that asks the user to enter an angle in degrees and prints the sine of the angle. The \texttt{sin()} function is included in the standard math library.

12. Write a Fortran program that asks the user to enter an angle in degrees and prints the sine of the angle.

\section*{19.5 Comments}

Comments are critical for making computer programs readable by humans. Good programmers do their best to make the code itself as readable as possible, thus reducing the number of comments needed, but comments are always necessary to clarify.

Keep in mind that a computer program is a way to explain something to a computer, which requires a different approach and far more detail than human beings are used to dealing with. Hence, it’s impossible to make the code itself completely self-documenting.

In ANSI C and C++, a comment is everything from a // to the end of the line, or everything between /* and */, even if it spans lines.

In Fortran 90 and later, a \texttt{comment} is everything from a ’!’ character to the end of the line.

Well-written programs are typically around 1/3 comments, 2/3 code, but this varies widely depending on the complexity of the program. Simple programs may require fewer comments, and complex programs may require more comments than code.

\begin{center}
\begin{tabular}{c}
\textbf{Caution} \\
Good comments build good karma. \\
If you neglect to comment your code, you will have a hard time understanding it a few days later! \\
Furthermore, other programmers will not be able to understand it, and will say mean things about you.
\end{tabular}
\end{center}

\subsection*{19.5.1 Block Comments}

Block comments are multi-line comments that document a section of code, such as a whole subprogram, a loop, or just a block of code that does a particular task. They are formatted nicely and consistently so that they stand out, and are separated from the other code by blank lines above and below.

\begin{verbatim}
/*
 * Description:
 * Compute the area of a circle given the radius as input.
 */

!-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-
! Description:
! Compute the area of a circle given the radius as input.
!-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-
\end{verbatim}
19.5.2 Line Comments

Line comments document one to a few lines of code, and may appear above the code or to the right of it. If above the code, they should not be separated from the code they document with a blank line, but should be separated from the code above. If next to the code, they should be indented consistently with other line comments to make the code easy to read.

```fortran
double radius, // Radius of the circle
     area;    // Area of the circle

// Input the radius
printf("What is the radius of the circle? ");
scanf("%lf", &radius);

use ISO_FORTRAN_ENV ! Enable INPUT_UNIT, OUTPUT_UNIT, ERROR_UNIT, etc.
! Disable implicit definitions (i-n rule)
implicit none
```

19.5.3 Self-test

1. What is the purpose of comments?
2. How are comments written in Fortran? In C?
3. Where do we use block comments?
4. Where do we use line comments?

19.6 Strings

A string is any sequence of characters such as letters, digits, and punctuation symbols. Most programming languages support the use of string constants, which consist of literal characters between quotes. C uses double quotes to enclose a string:

"What is the radius of the circle?"

Fortran uses single quotes to enclose a string:

'What is the radius of the circle?'

In C, we can enclose a single character between single quotes:

`putchar('x');`

A character between single quotes in C is not a string, however, but a single character, which is a different data type that cannot be used where strings are used.

19.6.1 Self-test

1. How do we write a string constant in Fortran? In C?
2. Do the type of quotes used make a difference in C? Explain.
19.7 The C Preprocessor

The C preprocessor, also used with C++ and Fortran, is a stream editor specially designed to modify source code. It recognizes language tokens, such as variable names, numeric constants, strings, etc.

Preprocessing was added to Fortran by developers who found the features useful in C. The Fortran preprocessor is based on the C preprocessor and is actually the same program in some cases.

Initially, programmers simply ran their Fortran code through cpp manually before running the Fortran compiler. With some modern compilers, such as the GNU Fortran compiler, the preprocessor is automatically invoked if the file name extension is capitalized (`.F90` vs `.f90`). This can only work, of course, if the file system is case-preserving. If the file name extension is not capitalized, then the preprocessor must be invoked manually prior to compilation.

Preprocessor actions are designated by directives, which are indicated by a `'#'` as the first character on the line. (The preprocessor is line-oriented, unlike the C and C++ compilers, which are completely free-format.)

19.7.1 `#include`

The `#include` directive inserts the contents of another source file into the stream at the point where the `#include` appears. The files included are known as header files. The use a file name extension of `.h` for C and `.hpp` for C++. They contain things like constant definitions, type definitions, and prototypes, which may be used by many other source files.

Factoring out commonly used code in this way eliminates redundancy and greatly reduces the maintenance cost of the source code.

Caution It is widely regarded as a very bad practice to place any executable statements in header files. Function definitions to be shared by multiple programs should be precompiled and saved in libraries, archives of object files rather than included from header files.

Header files provided by the system or installed globally with libraries have their names enclosed in angle brackets. The preprocessor will look for these headers in `/usr/include` and directories specified with the `-I` compiler flag.

For example, to include `/usr/include/stdio.h` and `/usr/include/sys/types.h` in your program, simply use the following:

```c
#include <stdio.h>
#include <sys/types.h>

cc myprog.c -o myprog
```

The `-I` flag indicates additional directories to check for headers. The path indicated in `#include` is relative to the prefix specified with `-I`.

For example, if you are using vector functions from the GNU Scientific Library (GSL) and the GSL headers are installed in `/usr/local/include/gsl`, then you could do the following:

```c
#include <gsl/gsl_vector.h>

cc -I/usr/local/include myprog.c -o myprog
```

Alternatively, you could use the following:

```c
#include <gsl_vector.h>

cc -I/usr/local/include/gsl myprog.c -o myprog
```

Header files that are part of the project and reside in the same directory as the code being compiled are enclosed in double quotes.

```c
#include "myheader.h"
```
19.7.2 #define

The C preprocessor is used to define named constants. Such constants have no type, since the names are replaced by their value with a simple text substitution.

The #define directive is followed by an identifier that must follow the same naming conventions as a C variable, i.e. it must begin with a letter or underscore, and subsequent characters can be letters, underscores, or digits.

Note Constant names defined with #define typically use all capital letters, so that they can be easily distinguished from variables where they are used in statements.

```
#define PI 3.14159265358979323846
#define RADIUS_PROMPT "Please enter the radius: 
```

Constants are actually the simplest case of what #define can be used for. It can also be used to define macros, which are an alternative to functions with some interesting capabilities and caveats. Macros are discussed in Section 24.14.

19.7.3 Self-test

1. What is special about the C preprocessor compared to other stream editors? What is an example of another stream editor that you know of?
2. What is a preprocessor directive? List two examples.
3. What is the difference between angle brackets and quotes used with #include?
4. Show you you would include the system header file stdlib.h in a C program.
5. Show how you would define the constant e, the natural number, in C. Might this constant already be defined for us somewhere? If so, where?
6. Does it matter whether we use upper or lower case letters with constants? Why or why not?

19.8 The Basics of Variable Definitions

A variable in a computer program is a name for a memory location, where we can store information such as integers, real numbers, and characters. See Section 18.5 for an explanation of computer memory.

A Variable definition allocates one or more memory cells for storing data, and binds a data type and a name to the memory address. High-level language programmers generally don’t know or care what the actual memory address is. We use variable names to refer to data stored in memory.

Note In C, a definition and a declaration are not the same thing. The latter does not allocate space, but only alludes to something defined elsewhere, to provide information about data types. This will be clarified later when we discuss various types of definitions and declarations. To avoid confusion, we will adhere to these terms throughout this text when discussing any language.

Variable definitions in C:

```
// Variable definitions for the main program
double radius, area;
```

Variable definitions in Fortran:
! Variable definitions for main program

double precision :: radius, area

The compiler keeps track of the location and data type of each variable, so when it encounters a variable name in a statement, it uses the correct memory address and binary format (character, integer, real number, etc.) to manipulate the data.

Caution
The original Fortran language included a rule known as the I through N rule, which allowed programmers to omit variable definitions, and allow the compiler to define them implicitly. The rule states that if a variable is not defined explicitly, and the variable’s name begins with any letter from I to N, then it is implicitly defined as an integer. Otherwise, it is implicitly defined as a real.

The rule was created in the days when programs were stored on paper punch cards and each line of code meant another punch card, so eliminating variable definitions saved a lot of work and paper. On today’s computers, there is no reason to keep such a rule, but nevertheless, Fortran has maintained it so that old code will still compile.

If you forget to define a variable explicitly, the Fortran compiler will quietly define it for you using the I-N rule, and often not with the data type you need. Furthermore, the I-N rule interferes with selection of descriptive variable names. The I-N rule can and should be disabled in all new Fortran code by adding the line

```
implicit none
```

above the variable definitions in every subprogram, including the main program. This will prevent variables from being accidentally defined with the wrong type.

Variable definitions are discussed in greater detail in Section 20.5.

19.8.1 Self-test

1. What is a variable?

2. Show a variable definition for a double precision variable called volume in C and in Fortran.

3. Explain the I-N rule.

4. What kinds of problems can the I-N rule cause? How can they be avoided?

19.9 C Statements

Unlike Fortran, the C language makes no distinction between assignment statements and subprogram calls.

In C, a statement other than a flow control statement is any expression followed by a semicolon. All of the following are valid statements as far as a C compiler is concerned:

```
area = M_PI * radius * radius;
printf("The area is %f.\n", area);
5;
area + 2;
```

The last two statements above don’t accomplish anything, of course, but are nevertheless perfectly valid. The C compiler may issue a warning about a statement with no effect, but the program will compile and run.

Part of the design philosophy behind the C language was "trust the programmer". Hence, C has an absolutely minimal number of rules. It does not forbid anything unless it makes it impossible to determine the meaning of the code.

Trusting the programmer also makes it easier for the programmer to do certain things. Many languages that came before C were considered somewhat obstructive.
This makes the C compiler much simpler and faster, and programmers tend to learn from their mistakes anyway, so it isn’t necessary for the compiler to contain extra code to police them.

Every expression in a C program has a value, including the call to the printf() function, which returns the number of characters printed, and the assignment statement above, which has a value of the number assigned. We are simply free to ignore the value if we choose.

The fact that an assignment statement is an expression with a value allows us to string assignments together:

\[ a = b = c = 10; \]

The expression \( x = 10 \) has a value of 10, which is assigned to \( b \), and so on.

Fortran does not allow this, since an assignment statement has a specific syntax of:

\[ \text{variable} = \text{expression} \]

19.9.1 Self-test

1. What is the difference between C statements and Fortran statements?
2. Why did the designers of C follow the "trust the programmer" philosophy?
3. What makes it possible to string together assignment expressions in C?
4. Write a C program that asks the user to enter a mass, \( m \), and calculates energy using Einstein’s equation \( E = mc^2 \). The program should ask the user to input \( m \), define a constant called \( \text{LIGHT\_SPEED} \) for \( c \) and output \( E \).

19.10 Fortran Statements

A statement is anything in a program that actually does some sort of work at run-time. There are three types of statements in Fortran, described below.

19.10.1 Subroutine Calls

A subroutine call statement jumps to another block of code, which could be part of the program, part of a library of subprograms stored in another file, or could be built into the Fortran language. Some commonly used intrinsic subroutines include:

- read: Inputs data from the standard input or any other input stream.
- write: Outputs data to the standard output, or any other specified output stream.
- print: Outputs data to the standard output stream.

\[ \text{print *}, 'What is the radius of the circle?' \]

19.10.2 Assignment Statements

An assignment statement assigns a new value to a variable, overwriting what was previously contained at that memory address.

\[ \text{area} = \text{PI} \times \text{radius} \times \text{radius} \]
19.10.3 Flow-control Statements

Statements in a Fortran program are normally executed in the order they appear. Flow-control statements, such as the if-then-else-endif in the sample program, alter the flow of statements based on some condition determined at run-time, such as whether the radius is positive or negative.

```fortran
if ( radius >= 0.0d0 ) then
   area = PI * radius * radius
   print *, 'The area is ', area
else
   print *, 'The radius cannot be negative.'
endif
```

Flow-control statements are covered in detail in Chapter 22 and Chapter 23.

19.10.4 Self-test

1. What are the three kinds of statements in a Fortran program?
2. What does a subroutine call do?
3. What does an assignment statement do?
4. What does a flow-control statement do?

19.11 Fortran Modules

Fortran modules are used to define things for use in more than one subprogram (including the main program). Each module has a name, and a subprogram can gain access to items defined in that module with a `use` statement.

```fortran
module constants
    ! Define ONLY constants here, not variables!
    ! (use the 'parameter' keyword)
    double precision, parameter :: 
        PI = 3.1415926535897932d0, 
        E = 2.7182818284590452d0, 
        TOLERANCE = 0.00000000001d0, 
        AVOGADRO = 6.0221415d23 ! Not known to more digits than this
end module constants

! Main program body
program example
    use constants  ! Constants defined above
    ...
end program
```

Caution

Modules should be used only to define constants, not variables. Hence, each definition should include the `parameter` modifier. Defining variables that can be modified by more than one subprogram will cause side effects. A side effect occurs when one subprogram modifies a variable and that change impacts the behavior of another subprogram. Issues caused by side effects are extremely difficult to debug, because it is impossible to tell by looking at a subprogram call what side effects it may cause. Hence, subprograms should only modify the values of variables that are passed to it as arguments, since these are visible in the call. Subprograms are described in detail in Chapter 24.
19.11 Self-test

1. What is the purpose of Fortran modules?
2. Why is it dangerous to define variables in a Fortran module?
3. Write a Fortran program that calculates energy using Einstein’s equation \( E = mc^2 \). The program should ask the user to input \( m \), define a constant called LIGHT_SPEED for \( c \) in a module called constants, and output \( E \).

19.12 Fortran Intrinsic Functions and the C Standard Libraries

19.12.1 C

In contrast, C has no intrinsic functions. All functions in C are separate from the compiler. The interface (required arguments and return values) of common functions such as `printf()`, `sin()`, `cos()`, `exp()`, etc. are specified in header files such as `stdio.h` and `math.h` and the functions themselves are provided by precompiled libraries such as `libc` and `libm`.

To use `printf()` in a C program, we must have `#include <stdio.h>`. To use `sin()` and other math functions, we must have `#include <math.h>` and compile with `-lm` to link with the standard math library.

19.12.2 Self-test

1. What is the difference between a Fortran intrinsic function and a library function like those used in C?
2. Write a Fortran program that asks the user for an angle in degrees and prints the sine of the angle.
3. Write a C program that asks the user for an angle in degrees and prints the sine of the angle.

19.12.3 Fortran

An intrinsic, or built-in function is a function that is recognized by the compiler.

Fortran provides a wealth of intrinsic mathematical functions for computing trigonometry functions, logarithms, etc. The complete list of functions is too extensive to list here.

Functions are used within expressions, and for the most part, look like they would in any algebraic expression. The general form is

\[ \text{name}(\text{argument }, \text{argument }...] \]

For example, the following code uses the `sin()` and `sqrt()` functions:

\[
\begin{align*}
y &= \sin(x) \\
\text{root1} &= (-b + \sqrt{b^2 - 4.0 \times a \times c}) / 2.0 \times a
\end{align*}
\]

In the code above, \( x \) is the argument to the `sin()` function, and \( b^2 - 4.0d0 \times a \times c \) is the argument to the `sqrt()` function.

Some functions return a specific type, while others are polymorphic, meaning that the same function returns a different type depending on the argument(s) it receives. Examples include the `anint()` rounding function, `sin()`, `cos()`, and `tan()`. Each of these functions returns a real if the argument is real, a complex if the argument is complex, etc.

Many functions can work with both real and complex values (i.e. real, double precision, complex, and double complex). Certain functions, by their mathematical nature, require the use of complex values for some cases. For instance, `sqrt(-1.0d0)` will cause an error since -1.0d0 is a real number, and no real square root exists for it. On the other hand, `sqrt((-1.0d0, 0.0d0))` will compute a valid result, since (-1.0d0, 0.0d0) is a complex number, and a complex square root does exist.
19.13 Code Quality

A program that works is not necessarily a good program. In fact, most programs that work are actually pretty crappy in most respects. There are several objective and subjective measures of good code. Most code quality measures and advice are specific to particular topics in programming and are therefore discussed throughout the text as new topics are presented. Below is a brief introduction to the basic concepts.

There are tools available to help you check code quality. The GCC and Clang compilers both provide a `-Wall` flag to issue as many warnings as possible about potential issues during compilation.

Most Unix systems provide the `lint` command for further checking C code for potential issues. (The command is so named because it picks the "lint" off your source code.)

There are similar source code analysis tools for most other languages as well.

- Readability is a somewhat subjective measure of how easy it is for people to understand the code. Readability is determined by several factors such as comments that explain why the code is doing what it’s doing, variable names that make it clear what a variable contains, etc.
  
  Good code format involves consistent indentation, spacing (blank lines) to separate logical sections of code that perform separate tasks.

- Speed is an objective measure that refers to the how long the program takes to perform a task. Naturally, we want most programs to run as fast as possible. Tips for speeding up code are presented throughout the text.

- Resource requirements are another objective measure that refers to how much memory and disk space a program requires to run. Programs that use more than they should need to in theory are referred to as bloatware.

- Robustness is an objective measure that refers to the ability of a program to handle user errors and other bad input. The worst case with any program is incorrect output. This should simply never happen. The best case is a program that detects all bad input and takes appropriate action such as telling the user exactly what they need to do to correct it.

- Add more...

Most existing programs could be made to run much faster and use far fewer resources. Most programmers are content if the program works and is fast enough for their purposes. This is sometimes OK, but can create problems when someone wants to run the program with bigger inputs or on a computer with less memory or a slower CPU.

Hence, it’s best to try to write the fastest code possible and minimize resources used every time you write new code.

19.13.1 Self-test

1. Describe two subjective aspects of code quality.

2. Describe two objective aspects of code quality.

19.14 Performance

At this point in the book, there isn’t a whole lot to say about performance, except that you should try to maximize it at all times.

While a program may be "fast enough" for your current needs on your current computer, you may later need to run it using much larger inputs or on a slower computer with less memory.

Maximizing performance is a matter of choosing the best algorithms, the best programming language, and keeping the execution path (the number of statements executed in order to complete a task) as short as possible.

Generally, shorter code is faster code, so set out from the start to learn how to write the most concise and efficient code possible.
19.14.1 Code Examples

19.14.2 Self-test

1. Why should we always try to maximize the performance of our programs?
Chapter 20

Data Types

20.1 Internal Representation

The data types offered by C and Fortran generally correspond to what typical hardware provides, such as 8, 16, 32, and 64-bit integers, and 32 and 64-bit floating point numbers.

Most compilers also support 64-bit integers on 32-bit CPUs, in which case operations like addition and subtraction will take twice as long, since the CPU has to do it with two 32-bit machine instructions. Likewise, most compilers support 128-bit integers and floating point values, which will require multiple machine instructions on 64-bit hardware.

Integers are usually represented internally in unsigned binary or two’s complement signed integer format, since these formats are directly supported by most hardware.

Floating point types (real, real(8), complex) are usually represented in IEEE floating point format, which is the standard supported by most hardware. There are some CPUs that do not have floating point support at the hardware level, but such CPUs are only generally used in embedded applications and not for scientific computing.

Character types are generally processed internally as 8-bit unsigned integers representing the binary ISO character code. Some systems support 16 and 32-bit ISO character sets as well, depending on the locality selected in the operating system configuration. The 16 and 32-bit codes are only needed for non-alphabetic languages such as Chinese.

20.2 Standard C Types

C supports the typical data types supported directly by most CPUs, as well as some extended types. Table 20.1 outlines the standard data types available in C.

C integer types can be prefixed with the modifier unsigned, which doubles the positive range and eliminates the ability to represent negative values.

Note that C’s int and long types vary in size from one platform to another. Because of this, we have to make pessimistic assumptions about the range of these types in order to write portable code. For example, we must assume that an int has a maximum value of +32,767, even though it can hold up to +2,147,483,647 on most platforms. Likewise, we have to assume a long is limited to +2,147,483,647, even though it can go to much larger values on 64-bit platforms.

Note: The size of int and long can be controlled by compiler flags in most cases. See your compiler documentation for details.

The int and long types are provided for the sake of speed. An int is always the fastest integer type on a given platform, so if you want speed and don’t care about range or how much memory it uses, then use int.
<table>
<thead>
<tr>
<th>C Type</th>
<th>Description</th>
<th>Range</th>
<th>Precision</th>
</tr>
</thead>
<tbody>
<tr>
<td>char</td>
<td>8-bit signed integer</td>
<td>-128 to +127</td>
<td>Exact</td>
</tr>
<tr>
<td>short</td>
<td>16-bit signed integer</td>
<td>-32,768 to +32,767</td>
<td>Exact</td>
</tr>
<tr>
<td>int</td>
<td>16 or 32-bit signed integer</td>
<td>-32,768 to +32,767 or -2,147,483,648 to +2,147,483,647</td>
<td>Exact</td>
</tr>
<tr>
<td>long</td>
<td>32 or 64-bit signed integer</td>
<td>-2,147,483,648 to +2,147,483,647 or +/- 9.22337203685e+18</td>
<td>Exact</td>
</tr>
<tr>
<td>long long</td>
<td>64 or 128-bit signed integer</td>
<td>+/- 9.22337203685e+18 or +/- 1.7014118346e+38</td>
<td>Exact</td>
</tr>
<tr>
<td>unsigned char</td>
<td>8-bit unsigned integer</td>
<td>0 to 255</td>
<td>Exact</td>
</tr>
<tr>
<td>unsigned short</td>
<td>16-bit unsigned integer</td>
<td>0 to 65,535 or 4,294,967,295</td>
<td>Exact</td>
</tr>
<tr>
<td>unsigned int</td>
<td>16 or 32-bit unsigned integer</td>
<td>0 to 65,535 or 4,294,967,295</td>
<td>Exact</td>
</tr>
<tr>
<td>unsigned long</td>
<td>32 or 64-bit unsigned integer</td>
<td>0 to 4,294,967,295 or 1.84467440737e+19</td>
<td>Exact</td>
</tr>
<tr>
<td>unsigned long long</td>
<td>64 or 128-bit unsigned integer</td>
<td>0 to 1.84467440737e+19 or 3.40282366921e+38</td>
<td>Exact</td>
</tr>
<tr>
<td>float</td>
<td>32-bit floating point</td>
<td>+/- (1.1754 x 10^-38 to 3.4028 x 10^38)</td>
<td>24 bits (6-7 decimal digits)</td>
</tr>
<tr>
<td>double</td>
<td>64-bit floating point</td>
<td>+/- (2.2250 x 10^-308 to 1.7976 x 10^308)</td>
<td>52 bits (15-16 decimal digits)</td>
</tr>
<tr>
<td>long double</td>
<td>128-bit floating point</td>
<td>+/- 3.3621 x 10^-4932 to 1.1897 x 10^+4932</td>
<td>114 bits (64 decimal digits)</td>
</tr>
<tr>
<td>float complex</td>
<td>Two 32-bit floating point values</td>
<td>Same as float</td>
<td>Same as float</td>
</tr>
<tr>
<td>double complex</td>
<td>Two 64-bit floating point values</td>
<td>Same as double</td>
<td>Same as double</td>
</tr>
<tr>
<td>long double complex</td>
<td>Two 128-bit floating point values</td>
<td>Same as long double</td>
<td>Same as long double</td>
</tr>
</tbody>
</table>

Table 20.1: C Data Types
If you need integers larger than +32,767, then use long, but be aware that it may be a multiple-precision data type on 16-bit CPUs, in which case it will be much slower.

If you don’t need numbers larger than +32,767 and you want to limit the size of the variable to 2 bytes, then use short. This is typically only done on embedded systems with very little memory or when using large arrays. Note that short values will be promoted to int in many situations, which will slow down the program.

If you need to use integer types with a specific size and range, there are several data types such as int32_t and uint32_t, as well as additional macros and functions for using them defined in inttypes.h. These types are not part of the C language, but C makes it easy to define new data types, so there are many derived data types available in the header files.

```c
#include <inttypes.h>

int main(int argc, char *argv[])
{
    int32_t myint;
}
```

Logical/Boolean values in C do not have a separate data type. Instead, C treats them as they are handled internally, as integers. A value of 0 represents false, and any non-zero value represents true.

The standard header file /usr/include/stdbool.h defines a data type called bool and constants true and false.

Finally, the integer type size_t is defined in the standard header files for use as an array subscript. Arrays are discussed in Chapter 26. The size_t type is an unsigned integer of the same size as a memory address on the underlying hardware, so it is the right size to subscript any array. It is defined in stddef.h, which is included from most other standard header files, so it need not be included explicitly in your program.

### 20.2.1 Self-test

1. What would be the best C data type to use for each of the following values? Explain your reasoning in each case.
   (a) A person’s age, in years.
   (b) The temperature of a star, up to 40,000 Kelvin.
   (c) The balance of a checking account, in pennies.
   (d) The federal debt, in dollars.
   (e) Avogadro’s number.

### 20.3 Standard Fortran Types

Fortran provides all the typical data types supported directly by most CPUs, as well as a few abstract types. Table 20.2 outlines the standard data types available in Fortran 90.

#### 20.3.1 Self-test

1. What would be the best Fortran data type to use for each of the following values? Explain your reasoning in each case.
   (a) A person’s age, in years.
   (b) The temperature of a star, up to 40,000 Kelvin.
   (c) The balance of a checking account, in pennies.
   (d) The federal debt, in dollars.
   (e) Avogadro’s number.
### Fortran 90 Data Types

<table>
<thead>
<tr>
<th>Fortran 90 Type</th>
<th>Description</th>
<th>Range</th>
<th>Precision</th>
</tr>
</thead>
<tbody>
<tr>
<td>integer(1)</td>
<td>8-bit signed integer</td>
<td>-128 to +127</td>
<td>Exact</td>
</tr>
<tr>
<td>integer(2)</td>
<td>16-bit signed integer</td>
<td>-32,768 to +32,767</td>
<td>Exact</td>
</tr>
<tr>
<td>integer(4)</td>
<td>32-bit signed integer</td>
<td>-2,147,483,648 to +2,147,483,647</td>
<td>Exact</td>
</tr>
<tr>
<td>integer(8)</td>
<td>64-bit signed integer</td>
<td>+/- 9.22 x 10^18</td>
<td>Exact</td>
</tr>
<tr>
<td>integer(16)</td>
<td>128-bit signed integer</td>
<td>+/- 9.22 x 10^18 or +/- 1.70 x 10^38</td>
<td>Exact</td>
</tr>
<tr>
<td>real(4)</td>
<td>32-bit floating point</td>
<td>+/- (1.1754 x 10^-38 to 3.4028 x 10^38)</td>
<td>24 bits (6-7 decimal digits)</td>
</tr>
<tr>
<td>real(8)</td>
<td>64-bit floating point</td>
<td>+/- (2.2250 x 10^-308 to 1.7976 x 10^308)</td>
<td>52 bits (15-16 decimal digits)</td>
</tr>
<tr>
<td>real(16)</td>
<td>128-bit floating point</td>
<td>+/- 3.3621 x 10^-4932 to 1.1897 x 10^4932</td>
<td>114 bits (64 decimal digits)</td>
</tr>
<tr>
<td>character</td>
<td>8-bit ISO, 16-bit in non-Latin locales</td>
<td>ISO 0 (NUL) to 255 (y-umlaut in ISO-Latin1)</td>
<td>Exact</td>
</tr>
<tr>
<td>logical</td>
<td>.true. or .false.</td>
<td>false to true</td>
<td>Exact</td>
</tr>
<tr>
<td>complex(4)</td>
<td>Two 32-bit floating point values</td>
<td>Same as real(4)</td>
<td>Same as real(4)</td>
</tr>
<tr>
<td>complex(8)</td>
<td>Two 64-bit floating point values</td>
<td>Same as real(8)</td>
<td>Same as real(8)</td>
</tr>
<tr>
<td>complex(16)</td>
<td>Two 128-bit floating point values</td>
<td>Same as real(16)</td>
<td>Same as real(16)</td>
</tr>
</tbody>
</table>

Table 20.2: Fortran 90 Data Types

### 20.4 The Cost of Multiple Precision

Most high level languages allow us to define integer and/or floating point variables that are larger than the maximum size supported by the CPU.

For example, in C, we can define variables of type `long long`, which is typically a 64-bit integer on 32-bit processors, or a 128-bit integer on 64-bit processors.

Some programs, such as the Unix `bc` command, and some libraries used in languages like C and Fortran, offer **arbitrary precision** variables and operations, where the size of a variable and the number of significant figures is limited only by the amount of memory available.

A 32-bit CPU has machine instructions that can perform addition, subtraction, etc. on 32-bit integers. Typically, they can also perform 8-bit and 16-bit integer operations with a single instruction.

When using 64-bit integers on a 32-bit computer, or 32-bit integers on a 16-bit computer, the CPU can only process half of the bits with each machine instruction, so it takes at least two instruction cycles, or twice as long, to complete these operations. This is known as a **multiple precision** operation. Usually, addition, subtraction, and several other basic operations on a 64-bit integer can be done with just two 32-bit instructions. Other multiple precision operations such as integer multiplication and any multiple-precision floating point operation may take much longer.

Hence, if speed is important, we want to avoid using data types that will require multiple precision operations on any computer likely to run the program.

Most PCs in use since the early 1990s support up to 32-bit integers and up to 64-bit floating point values. Hence, these data types are very safe to use in any typical PC application, and will only have performance issues if used on a 16-bit or smaller CPU, which is only likely to be found in embedded applications such as under the hood of your car or in your washing machine. Be aware that much if the code you write in C could end up being used on anything from an embedded CPU to a large supercomputer, so do not make assumptions about hardware when you code.
20.4.1 Self-test

1. Explain why multiple-precision arithmetic takes longer than single-precision.

2. What can we do to avoid multiple-precision operations?

20.5 Variable Definitions

Variable definitions were introduced in Section 19.8. We will now examine them in greater detail.

A variable definition allocates memory to store a value, and assigns the memory location a name and a data type. The compiler uses the data type to determine which machine instructions to use to process the data. For example, adding two integers is done with a different machine instruction than adding two floating point values.

A C variable definition consists of a data type followed by one or more variable names separated by commas, and ultimately a semicolon:

```
type name [, name ...];
```

```double height, width, area;```

A Fortran 90 variable definition consists of a type, followed by two colons, followed by a list of variable names separated by commas:

```
type :: name [, name ...]
```

```real(8) :: height, width, area```

Optional sizes in () may follow many types, and indicate the number of bytes. For numeric types, the size is usually 4 or 8 bytes (32 or 64 bits), since these are the integer and floating point sizes supported by most hardware. Use of other sizes should be avoided for the sake of portability and speed.

Older Fortran 77 programs may use a “*” instead of () to indicate non-default data type sizes. This syntax is obsolete and should not be used in new code.

```integer*8 list_size```

Variable names must begin with a letter or an underscore, and may contain letters, underscores, and digits.

---

**Caution** The words “definition” and “declaration” are often used interchangeably in programming. In the context of languages such as C and C++, definition and declaration have different meanings. (A definition in C allocates memory, whereas a declaration merely alludes to a variable or function defined elsewhere.)

20.5.1 Initializers

Both C and Fortran support assigning an initial value to a variable in the definition.

```double sum = 0.0;
```

```real(8) :: sum = 0.0d0```

Using this feature reduces the length of a program slightly by eliminating an assignment statement further down.

Some would argue that it is less cohesive, however. It is generally a good practice to group related statements together in one cohesive block. In this context, it would mean initializing variables near the code that depends on that initialization. This way, we can see that the variable is initialized properly without having to scroll or search through the source code. This will save time and distractions while debugging.
double sum;

// Some other block of code

// A cohesive block of code
sum = 0.0;
for (c = 0; c < list_size; ++c)
    sum += value;

double sum = 0.0;

// Suppose there’s 100 lines of additional code here

// This is not cohesive since the initialization of sum is separated from
// the loop
for (c = 0; c < list_size; ++c)
    sum += value;

**Note** Variables are initialized at the time they are created. This has different connotations, depending on storage class, which is discussed in Section 24.11.

### 20.5.2 Memory Maps

Memory locations for variables are generally allocated together in a block. Suppose we have the following variable definitions:

double height, width, area;
int list_size, age;

or

real(8) :: height, width, area
integer :: list_size, age

The program will have 8 bytes (64 bits) of memory allocated for each double or real(8) variable, and 4 bytes (32 bits) for each integer variable. The C int variables will also require 4 bytes on most systems. A possible map of the memory space, assuming the block begins at memory address 4000, is shown in Table 20.3. Recall that each memory location contains 1 byte (8 bits), so each of these variables will occupy multiple memory addresses.

<table>
<thead>
<tr>
<th>Address</th>
<th>Variable</th>
</tr>
</thead>
<tbody>
<tr>
<td>4000 to 4007</td>
<td>height</td>
</tr>
<tr>
<td>4008 to 4015</td>
<td>width</td>
</tr>
<tr>
<td>4016 to 4023</td>
<td>area</td>
</tr>
<tr>
<td>4024 to 4027</td>
<td>list_size</td>
</tr>
<tr>
<td>4028 to 4031</td>
<td>age</td>
</tr>
</tbody>
</table>

Table 20.3: Memory Map

### 20.5.3 Self-test

1. State one argument for and one against using initializers in a variable definition.

2. Draw a possible memory map for the following variable definitions:

```c
int age, year;
float gpa;
```
20.6 Constants

20.6.1 Literal Constants

Like variables, C and Fortran constants also have types, which are determined by how the constant is written.

For example, constants containing a decimal point are floating point values. Whether they are single or double precision is determined by an optional suffix.

Table 20.4 and Table 20.6 illustrate the types that compilers assign to various constants.

<table>
<thead>
<tr>
<th>Constant</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>45</td>
<td>int</td>
</tr>
<tr>
<td>45u</td>
<td>unsigned int</td>
</tr>
<tr>
<td>-45</td>
<td>long</td>
</tr>
<tr>
<td>-45l</td>
<td>unsigned long</td>
</tr>
<tr>
<td>-45ll</td>
<td>long long</td>
</tr>
<tr>
<td>45ull</td>
<td>unsigned long long</td>
</tr>
<tr>
<td>4.5e1</td>
<td>double</td>
</tr>
<tr>
<td>4.5f</td>
<td>float</td>
</tr>
<tr>
<td>45.0l</td>
<td>long double</td>
</tr>
<tr>
<td>(45.0, 0.0)</td>
<td>double complex</td>
</tr>
<tr>
<td>'A'</td>
<td>int (not char!)</td>
</tr>
<tr>
<td>&quot;45&quot;</td>
<td>string (array of char)</td>
</tr>
</tbody>
</table>

Table 20.4: C Constants and Types

Note that a character between single quotes in C is not a string constant, but a character constant. It represents the ISO code of that character, and its type is int, not char. For example, 'A' is exactly the same as the integer constant 65, '0' is 48, and '! ' is 33.

Note In C++, the type of a character constant is char. This is one of very few points where C++ is not backward-compatible with C.

There are also special sequences known as escape sequences to represent non-graphic characters. The most common ones are listed in Table 20.5.

<table>
<thead>
<tr>
<th>Sequence</th>
<th>ISO code (decimal)</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>'n'</td>
<td>10</td>
<td>Newline / Line feed</td>
</tr>
<tr>
<td>'v'</td>
<td>13</td>
<td>Carriage return</td>
</tr>
<tr>
<td>'0'</td>
<td>0</td>
<td>Null byte</td>
</tr>
<tr>
<td>'t'</td>
<td>9</td>
<td>Tab</td>
</tr>
</tbody>
</table>

Table 20.5: C Escape Sequences

Self-test

1. Show how to write the constant 32 so that it has each of the following data types:
   
   (a) C: int, float, long long, double, float complex
   
   (b) Fortran: integer, real(8), complex, integer(2)
Table 20.6: Fortran 90 Constants and Types

### 20.6.2 Named Constants

Virtually every constant in your program should be assigned a name. Using names for constants makes the code much easier to read and allows the program to be more easily modified.

In C, we can define constants using the `#define` preprocessor directive, or using a variable definition with the `const` modifier.

```c
#define PI 3.1415926535897832

const double PI = 3.1415926535897832;
```

When using `#define`, the preprocessor simply replaces the name (PI) with the value (3.1415926535897832) wherever it is used as an identifier in the program. In the `printf` statement below, only the second instance of PI is replaced by the preprocessor.

```c
printf("The value of PI is ", PI);
```

In Fortran 90, constants are defined like variables, but with the `parameter` modifier, which informs the compiler that the variable is read-only. Any attempts to alter its value in the code will cause a compile error.

```fortran
real(8), parameter :: PI = 3.1415926535897832d0
```

A constant without a name is known as a hard-coded constant. Using named constants such as PI and MAX_RADIUS throughout the program instead of hard-coded constants such as 3.1415926535897832 and 10.0 has the following advantages:

- It allows the compiler to catch typos. If we mistype PI or MAX_RADIUS, the compiler will complain about an undefined variable or constant name. If we mistype 10.5 as 1.05, the program will compile just fine, but produce incorrect output. This is the hardest type of bug to find, and could have disastrous results if it goes undetected. Bugs like this have been known to cause catastrophic failures in cars, planes, spacecraft, and consumer products, leading to heavy financial losses, injury, and even death.

- If the names of the constants are descriptive (as they should be), the reader has an easier time understanding the code. It will likely take a bit of effort to figure out what a hard-coded constant actually means in a given context, whereas a good name makes it obvious.

- If you need to change the value of a constant (not an issue for PI, but quite possible for MAX_RADIUS), then having a named constant means you only need to make one change to the program. If you have the hard-coded constant 10.5 sprinkled all of the program, you’ll have to carefully hunt them all down and change them. You can’t simply change every instance of 10.5, since some of them might have a different purpose, and coincidentally have the same value as the maximum radius.

Example of bad C code:
Example of better C code:

```c
#include <stdio.h>
#include <math.h>  // Use M_PI
#include <sysexits.h>

#define MAX_RADIUS 10.0

int main(int argc, char *argv[]) {
    double radius;

    printf("Please enter the radius: ");
    scanf("%lf", &radius);
    if (radius < MAX_RADIUS) {
        printf("Area = %f\n", M_PI * radius * radius);
    } else {
        printf(stderr, "Radius cannot exceed %f.\n", MAX_RADIUS);
        return EX_OK;
    }
}
```

Example of bad Fortran code:

```fortran
program circle_area
    use iso_fortran_env
    implicit none
    real(8) :: radius

    print *, 'Please enter the radius:'
    read *, radius
    if (radius <= 10.0d0) then
        print *, 'Area = ', 3.1415926535897932d0 * radius * radius
    else
        write(ERROR_UNIT, '*') 'Radius cannot exceed 10.0.'
    endif
end program
```

Example of better Fortran code:

```fortran
module constants
```

```
! Define only constants in modules, not variables! (i.e. use ‘parameter’)
real(8), parameter :: &
   PI = 3.1415926535897932d0, &
   MAX_RADIUS = 10.0d0
end module constants

program circle_area
   use iso_fortran_env
   use constants ! Constants defined above
   implicit none
   real(8) :: radius
   print *, 'Please enter the radius:'
   read *, radius
   if ( radius <= MAX_RADIUS ) then
      print *, 'Area = ', PI * radius * radius
   else
      write(ERROR_UNIT, *) 'Radius cannot exceed', MAX_RADIUS
   endif
end program

Self-test
1. Show how to define the constant e, the natural number, as a 64-bit floating point value, 2 ways in C and one way in Fortran.

2. What is a hard-coded constant?

3. Explain three reasons not to use hard-coded constants in a program.

20.7 Choosing the Best Data Type

Choosing data types is important in order to maximize the speed of your programs, minimize memory use, and ensure correct results. The approach to choosing a data type is basically the same in any programming language. We simply need to understand the limitations of the data types offered by the language and how they are supported by typical hardware.

Choosing the best data type always involves understanding the needs of the specific code you are working on. There are, however a few general ideas that can be applied when trying to decide:

1. Use integers instead of floating point whenever possible. They are faster and more precise. Recall that floating point numbers are stored in a format like scientific notation. To add two scientific notation values requires three steps:

   (a) Equalize the exponents
   (b) Add the mantissas
   (c) Normalize the results

   Each of the three steps takes about as long as an integer addition (the second step essentially is an integer addition). Hence, we can expect floating point addition to take about three times as long as integer addition.

   Integers are more precise because all the bits are used to represent digits, whereas a floating point value with the same number of bits uses some of them for the exponent, which does not add precision. Only the bits used for the mantissa provide significant figures in our real numbers.

   We can often eliminate the need for floating point by simply choosing smaller units of measurement, so that we no longer have fractions in our data. For example, specifying monetary amounts in cents rather than dollars allows us to use integers:
```c
#include <stdio.h>
#include <sysexits.h>

int main(int argc, char *argv[]) {
    /* Use a type with enough range for intermediate results. */
    unsigned long long balance;

    printf("Enter the balance in pennies: ");
    scanf("%lld", &balance);

    /* Do multiplications first to avoid truncation */
    printf("Interest = %lld
", balance * 25 / 1000);

    /* Dividing first will not work */
    printf("Interest = %lld
", 25 / 1000 * balance);
    return EX_OK;
}
```

2. If you must use floating point, use a 64-bit floating point value unless you need to reduce memory use (i.e. your program uses large arrays or other in-memory data structures) and you do not need much precision. Modern computers do not take significantly longer to process 64-bit floating point values than they do to process 32-bit floating point values. Keep in mind that the precision of your results is usually less than the precision of the data type you use, since round-off error accumulates during calculations. 32-bit floating point operations are only accurate to 6 or 7 decimal digits, which could skew the results for some calculations.

3. Make use of complex data types where they can be helpful. Allowing computations to go into the complex plane can simplify your code. Note, however, that calculations with complex numbers take longer than the same calculations with real numbers.

4. Among integer types, chose the fastest type that provides the necessary range for your computations. In C, the `int` type is always the fastest data type for the underlying hardware. Note that it’s size will differ, depending on the CPU running the program. It may be as small as 16 bits, so if there is any chance that your code will be used on a 16-bit processor, you have to assume that `int` is limited to a range of -32,768 to +32,767, and `unsigned int` is limited to 0 to 65,536. Data types such as long and long long will provide greater range, but may require multiple precision arithmetic on some CPUs. For example, it will take two machine instructions to add 64-bit integers on a 32-bit CPU, so that operation will take twice as long as adding `int` values.

The `char` and `short` types can be used to save memory where we need a large array of integers within the range of those types. This can also slow down your program, though, since these types will often be promoted to `int` or larger data types when used in expressions. The conversion from 8-bit or 16-bit integers to 32-bit integers and back require additional machine instructions between useful instructions like add and multiply. This is discussed in greater detail in Section 20.9.

### 20.7.1 Self-test

1. Which is generally faster, integer types of floating point types? Explain why.
2. Which offers more precision, a 64-bit integer or a 64-bit floating point value? Explain.
3. When choosing a floating point type, when should you use single precision and when should you use double precision?
4. Explain one advantage and one disadvantage to using complex numbers in a program.
5. What is the range of each of the following data types?
(a) Fortran integer.
(b) Fortran integer(2).
(c) C int.
(d) C long.
(e) C long long.
(f) C uint32_t.

6. Which C integer type should we use for each of the following:

(a) A person’s age.
(b) A very large array of people’s ages.
(c) A temperature in Celsius for any point in the solar system.
(d) A temperature in Kelvin for any point in the solar system.

20.8 Math Operators and Expressions

C and Fortran supports mostly the same math operators, except that Fortran supports exponentiation, which is done in C using the pow() library function.

Precedence and order of evaluation is the same as in algebra. Grouping () has the highest precedence, followed by negation (unary minus), then multiplication and division, then addition and subtraction.

C also has a number of additional operators for bitwise operations, since it is often used for systems programming, cryptography, etc.

<table>
<thead>
<tr>
<th>Operator</th>
<th>Operation</th>
<th>Precedence</th>
<th>Order of Operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>( )</td>
<td>grouping</td>
<td>1 (highest)</td>
<td>inside to outside</td>
</tr>
<tr>
<td>-</td>
<td>negation</td>
<td>2</td>
<td>right to left</td>
</tr>
<tr>
<td>*</td>
<td>multiplication</td>
<td>3</td>
<td>left to right</td>
</tr>
<tr>
<td>/</td>
<td>division</td>
<td>3</td>
<td>left to right</td>
</tr>
<tr>
<td>+</td>
<td>addition</td>
<td>4</td>
<td>left to right</td>
</tr>
<tr>
<td>-</td>
<td>subtraction</td>
<td>4</td>
<td>left to right</td>
</tr>
<tr>
<td>&lt;&lt;</td>
<td>shift left</td>
<td>5</td>
<td>left to right</td>
</tr>
<tr>
<td>&gt;&gt;</td>
<td>shift right</td>
<td>5</td>
<td>left to right</td>
</tr>
</tbody>
</table>

Table 20.7: C Operators

The C shift operators can often be used in place of integer multiplication to greatly improve program speed. Note that when we shift the digits of a decimal number one place to the left, it has the effect of multiplying the value by 10. Likewise, shifting to the right divides by 10.

The same principal applies to any other number base, including base 2. This works with both unsigned binary and 2’s complement values. Hence, the following two statements are the same:

```
c = c * 2;
c = c << 1;
```

Depending on the capabilities of your CPU and your compiler’s optimizer, the second statement may be significantly faster. A shift of any number of bits can occur within a single clock cycle in most modern CPUs, while multiplication may require many clock cycles. Some CPUs can do integer multiplication very quickly and some optimizers are smart enough to recognize which of your multiplications can be done with a shift and will generate a shift instruction automatically. However, it’s best not to rely on such features. Using a shift instruction explicitly will ensure that your code is optimal no matter where it is compiled and executed.

The C increment and decrement operators are a convenient way to add 1 to or subtract 1 from a variable. The following two statements have the same effect:

```
c++;  
c = c + 1;
```
Like all C operators, increment and decrement operators can also be used within expressions, though. This often allows us to eliminate a separate statement for incrementing a counter variable, which improves code density.

When used this way, the value of the expression will depend on whether the ++ or -- comes before (pre-increment, pre-decrement) or after (post-increment, post-decrement) the variable. If it comes before, then the increment occurs before the value of the variable is used in the expression, and vice versa.

```c
a = 1;
b = 5 + ++a; // Pre-increment: b is now 7 and a is 2
```

```c
a = 1;
b = 5 + a++; // Post-increment: b is now 6 and a is 2
```

C also allows us to combine any binary (two-operand) operator with `=` to save typing in common situations. The following two statements are equivalent:

```c
c = c + 7;
c += 7;
```

Fortran supports most common algebraic operators, including exponentiation:

<table>
<thead>
<tr>
<th>Operator</th>
<th>Operation</th>
<th>Precedence</th>
<th>Order of Operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>()</td>
<td>grouping</td>
<td>1 (highest)</td>
<td>inside to outside</td>
</tr>
<tr>
<td>**</td>
<td>exponentiation</td>
<td>3</td>
<td>right to left</td>
</tr>
<tr>
<td>-</td>
<td>negation</td>
<td>3</td>
<td>right to left</td>
</tr>
<tr>
<td>*</td>
<td>multiplication</td>
<td>4</td>
<td>left to right</td>
</tr>
<tr>
<td>/</td>
<td>division</td>
<td>4</td>
<td>left to right</td>
</tr>
<tr>
<td>+</td>
<td>addition</td>
<td>5</td>
<td>left to right</td>
</tr>
<tr>
<td>-</td>
<td>subtraction</td>
<td>5</td>
<td>left to right</td>
</tr>
</tbody>
</table>

Table 20.8: Fortran Operators

**Example 20.1 Precedence and Order of Evaluation**

- `a + b + c` equals `(a + b) + c` equals `a + (b + c)`
- `a - b - c` equals `(a - b) - c` does not equal `a - (b - c)`
- `a ** b ** c` equals `a ** (b ** c)` does not equal `(a ** b) ** c`

Unary minus is the same as multiplication by -1, so `-x` is the same as `-1 * (-x)`, and `-x - 5` is the same as `-1 * x - 5.

Math operators are sensitive to the data types of the operands and may produce difference results for different types.
Caution
Recall from grade school that integer division and real division are not the same. Integer division results in an integer quotient and a remainder, while real division results in a different quotient and no remainder. (The real quotient is the integer quotient + remainder / divisor.)

The C and Fortran divide operator (/) performs integer division if both operands are integers, and floating point division if either operand is floating point. Pay attention to the types of the operands when dividing, whether they are variables or constants.

```c
int a = 5, b = 10;
double x, y, z = 10;
x = a / 2; // a and 2 are both integers, x = 2.0
y = a / 2.0; // 2.0 is floating point, y = 2.5
x = a / b; // a and b are both integers, x = 0.0
x = a / z; // z is floating point, x = 0.5
```

When using the division operator (/), care must be taken not to divide by zero.
Likewise, the exponentiation operator (** cannot be used with negative bases and certain fractional exponents, unless working with complex numbers. For example, (-1) ** (0.5) does not exist in the real numbers, and hence will cause an error in Fortran.

20.8.1 Self-test

1. What is the advantage of using C shift operators for multiplication and division?
2. Write a C and/or Fortran program that asks the user for the radius in meters and average density in of a planet in grams/milliliter, and prints the surface area, volume, and mass. Use the example programs at the beginning of Chapter 19 as models.
3. Write a C and/or Fortran program that asks the user for their bank account balance in pennies and prints the annual interest earned, also in pennies. The interest rate is 2.5%. All calculations should be done using only integers variables and constants. The program should be usable by Bill Gates. Use the example programs at the beginning of Chapter 19 as models.

20.9 Mixing Data Types

C and Fortran allow you to mix numeric data types in expressions. This should be avoided as much as possible, however, for two reasons:

- Most computer hardware cannot perform operations on two different data types. There are usually machine instructions for adding two integers of the same size, and machine instructions for adding two floating point values of the same size, but usually no machine instructions for adding an integer to a floating point value. Likewise, there are usually no instructions for adding two integers of different sizes or floating point numbers of different sizes.

  Before mixed data type operations occur, one value must be converted to the same type as the other. Some CPUs have instructions to convert between native data types. Other CPUs will require several instructions that manually manipulate the bits to perform a conversion. In either case, conversions cause a significant slow-down, as the compiler must insert additional machine instructions between the calculations to convert the values.

- Mixing data types also makes it more difficult to predict the output of a program, since it is easy to get confused and mistake integer operations for floating operations.
20.9.1 Implicit Data Conversions

Promotions occur when a two different data types are operands to a mathematical operator. In all cases, the value of the lower ranking type is converted to the higher ranking type.

The rules of promotion are as follows:

1. Complex types have the highest rank. Any other numeric type mixed with a complex value will be promoted to complex.
2. Floating point values have the next highest rank. Integers mixed with floating point values will be promoted to the floating point type.
3. Integers have the lowest rank and are always promoted to other types they are mixed with.
4. Within each general category of types (complex, floating point, and integer), a larger type ranks higher than a smaller one. E.g., a char will be promoted when mixed with any other integer, a short will be promoted when mixed with an int or larger, a C float / Fortran real(4) is promoted to double / real(8), and so on.
5. It is almost always a bad idea to mix signed and unsigned integers.

Demotions occur only when assigning an expression to a variable of a lower ranking type:

```fortran
integer :: area
real(8) :: height, width

area = height * width  ! Oops, lost the fractional part!
```

Note that when assigning any type of real value to an integer variable, the value is truncated, not rounded.

The type of a variable or constant is never changed. The promoted value is stored in a temporary memory location, and discarded when the program finishes evaluating the expression.

The examples below show all the steps necessary to evaluate a mixed expression, including calculations and implicit conversions. Note how implicit conversions can account for a significant fraction of the total operations.

Note also that some conversions, such as integer to floating point, can be rather expensive.

```fortran
int a, b, d;
double x;

a = 4;
b = 7;
x = 3.0;
d = b / a + x * 5.0 - 2.5;
```

```fortran
integer :: a, b, d
real(8) :: x

a = 4
b = 7
x = 3.0d0
d = b / a + x * 5.0 - 2.5
```
Assuming the CPU running this program can convert from one data type to another with a single instruction, this code would run significantly faster if all the variables and constants were the same data type to begin with. If the CPU cannot convert data types with a single instruction, this statement will be many times slower than necessary.

The compiler’s optimizer may be able to eliminate some promotions at run-time by performing them at compile-time. Nevertheless, mixing data types will usually slow down your code to some extent.

### 20.9.2 Explicit Data Conversions

In C, we can convert from one data type to another by simply prefixing the expression with the data type in parenthesis:

```c
int a = 1, b = 5;
double x;

// Convert value in a to double before division. This will cause an implicit promotion of the value of b when division occurs and produce a result of 0.5 instead of 0.
x = (double)a / b;
```

This is known as *casting* the expression.

Explicit data conversions can be performed in Fortran 90 using intrinsic functions. For example, to force a real division of two integer variables, we could do the following:

```fortran
integer :: a, b
real(8) :: x, y

y = dble(a) / dble(b) + x
```

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
<th>Returns</th>
</tr>
</thead>
<tbody>
<tr>
<td>int(A)</td>
<td>Truncated integer</td>
<td>integer</td>
</tr>
<tr>
<td>int2(A)</td>
<td>Truncated 16-bit integer</td>
<td>integer(2)</td>
</tr>
<tr>
<td>int8(A)</td>
<td>Truncated 64-bit integer</td>
<td>integer(8)</td>
</tr>
<tr>
<td>nint(A)</td>
<td>Nearest integer</td>
<td>integer</td>
</tr>
<tr>
<td>anint(A)</td>
<td>Nearest whole real/double</td>
<td>Same as A</td>
</tr>
<tr>
<td>real(A)</td>
<td>Nearest real</td>
<td>real</td>
</tr>
<tr>
<td>dble(A)</td>
<td>Nearest real(8)</td>
<td>real(8) [ double precision ]</td>
</tr>
<tr>
<td>cmplx(R [,I])</td>
<td>Nearest complex</td>
<td>complex</td>
</tr>
<tr>
<td>dcmplx(R [,I])</td>
<td>Nearest double complex</td>
<td>double complex [ complex(8) ]</td>
</tr>
</tbody>
</table>

Table 20.9: Explicit Conversion Functions

If you want to preserve the data type of a value and just round it, use anint():

```fortran
integer(8) :: a
real :: x

a = anint(x)
```
If you want to convert a rounded or truncated value to an integer, use int(), int2(), or int8().

20.9.3 Self-test

1. What is the output of the following code segment? (First try to determine it by reading the code, then type it in and run it to verify.)

```fortran
integer :: a, b, c
real(8) :: x, y
a = 1
b = 2
c = 3
x = 4.0d0
y = 5.0d0
print *, a / b
print *, 3 / 4
print *, c / 2
print *, x / y
print *, c / x
print *, x ** (1/2)
```

20.10 Code Quality

Code quality issues related to data types can be summarized as follows:

- Write literal constants so that they match the data type of other constants and variables used in the same expressions. This makes the code more self-documenting.
- Name all of your constants to make your code readable and modifiable.
- Avoid mixing data types in expressions as much as possible. Mixed may cause bugs that are difficult to track down.
- Don’t abbreviate variable and constant names. This is an irrational type of laziness that will save you a few seconds of typing and cost much more in debugging time later.

20.11 Performance

- Choose the fastest data type that has sufficient range and accuracy for your calculations.
- Write literal constants so that they match the data type of other constants and variables used in the same expressions. This will avoid causing promotions during evaluation of the expression.
- Avoid multiple-precision integer types. Operations on them take at least twice as long as the same operation on an integer type native to the underlying CPU.

20.11.1 Code Examples
Chapter 21

Basic Terminal Input/Output (I/O)

21.1 C Terminal Input/Output (I/O)

21.1.1 C Language Support

Odd as it may sound, the C language actually has no built-in I/O statements! The designers of the C language were careful not to include any features that could be implemented as subprograms. I/O fell into this category.

C I/O is performed by the C standard libraries, collections of subprograms that are written in C. The standard libraries include a large number of functions that perform I/O, manipulate character strings, perform common math functions, etc.

Recall from Section 7.4.3 that in Unix systems, all hardware access is ultimately performed by the kernel, the innermost layer of the operating system. The I/O functions in the C library simply call I/O functions in the kernel in order to perform basic I/O operations. There is no need for the C compiler to know what these functions are.

21.1.2 C Standard Streams

The C interfaces to the low level kernel I/O routines are provided by functions like open(), close(), read() and write(). The read() and write() functions simply read and write blocks of characters to or from an I/O device. This is the most direct and efficient way to perform input and output in bulk. It is the best approach for commands like cp, because many devices such as disks perform far better when reading or writing large blocks of data sequentially.

Stream I/O is another layer of software on top of the low level functions that allows us to conveniently input or output one character at a time. When we write a character to a stream, it is simply placed in a memory buffer (array), which is much faster than most I/O devices. When that buffer is full, it is dumped to the I/O device using a low-level write(). Likewise, a low-level read() is used to read a block of data from a device into a memory buffer, from which stream I/O functions can quickly fetch one character at a time.

We can use stream I/O to access any file or I/O device.

Recall from Table 7.8 that all Unix processes have three standard I/O streams from the moment they are born.

The C standard libraries provide a set of convenience functions specifically for reading from the standard input stream and writing to the standard output stream. The standard error stream is accessed using the more general functions used for any other stream.

Names for the standard streams are defined in the header file stdio.h, which we can incorporate into the program using:

```c
#include <stdio.h>
```
21.1.3 C Character I/O

The most basic C stream I/O functions are `getc()` and `putc()`, which read and write a single character.

```c
int ch;
ch = getc(stdin);
putc(ch, stdout);
```

**Note** The C language treats characters the same as integers. Hence, the `getc()` function returns an integer and the `putc()` function takes an integer as the first argument. More on this in Chapter 20.

All other stream I/O functions are built on top of `getc()` and `putc()`. You can also write additional stream I/O functions of your own using `getc()` and `putc()`.

The `getchar()` and `putchar()` functions are provided for convenience. They read and write a character from the standard input and standard output.

```c
int ch;
ch = getchar();
putchar(ch);
```

I.e., `getchar()` is equivalent to `getc(stdin)` and `putchar(ch)` is the same as `putc(ch, stdout)`.

21.1.4 C String I/O

The `puts()` and `gets()` functions read and write simple strings, which in C are arrays of characters, with a null byte (ISO character code 0, `\0`) marking the end of the content.

Arrays are discussed in Chapter 26. For now, we will only use simple examples necessary for understanding basic I/O.

```c
#define MAX_NAME_LEN 100;
char string[MAX_NAME_LEN+1];
puts("Please input your name: ");
gets(string);
puts(string);
```

**Caution**

The `gets()` function is considered dangerous, since it may input a string bigger than the array provided as an argument. This could lead to corruption of other variables in the program. Hence, other functions such as the more general `fgets()` should be used instead, even when reading from stdin.

```c
fgets(string, MAX_STRING_LEN, stdin);
```

Note that `fgets()` retains the newline (`\n`) character at the end of the string. If this is not desired, it must be removed manually after calling `fgets()`, or some other function should be used for input.
The `puts()` function appends a newline character to the string, so the next output will be on a new line. If this is not desired, one can use `fputs()` instead.

```c
fputs(string, stdout);
```

### 21.1.5 C Formatted I/O

The `printf()` and `scanf()` functions can be used for more complex input and output. Both `printf()` and `scanf()` require the first argument to be a *format string*, optionally followed by additional arguments that must match the format string. For each argument after the format string, the format string must contain a *placeholder* that matches the type of the argument.

```c
int fahrenheit = 76,
printf("The temperature is %d (%d Celsius)\n",
       fahrenheit, (fahrenheit - 32) * 5 / 9);
```

Some of the most common placeholders for `printf` are outlined in Table 21.2.

<table>
<thead>
<tr>
<th>Type</th>
<th>Number Format</th>
<th>Placeholder</th>
</tr>
</thead>
<tbody>
<tr>
<td>char, short, int</td>
<td>printable character</td>
<td>%c</td>
</tr>
<tr>
<td>string (character array)</td>
<td>printable characters</td>
<td>%s</td>
</tr>
<tr>
<td>char, short, int</td>
<td>decimal</td>
<td>%d</td>
</tr>
<tr>
<td>unsigned char, unsigned short, unsigned int</td>
<td>decimal</td>
<td>%u</td>
</tr>
<tr>
<td>size_t</td>
<td>decimal</td>
<td>%zu</td>
</tr>
<tr>
<td>float, double</td>
<td>decimal</td>
<td>%f</td>
</tr>
</tbody>
</table>

Table 21.2: Placeholders for `printf`

Prefixing any numeric placeholder with a lower case L (‘l’) corresponds to prefixing the type with ’long’. For example, `%ld` is for long int, `%lu` for unsigned long int, `%lld` for long long int, `%lf` for long double.

For complete information on available placeholders, run `man printf` or `man fprintf`.

**Note**

It may appear that the `printf` placeholders don’t always match the argument type, but this is because char, short, and float arguments are promoted to int or double when passed to `printf`. Because of this, `%f` is used for both float and double, and `%d` and `%c` can be used for char, short, and int alike.

The `scanf()` function reads formatted text input and converts the values to the proper binary format based on the placeholder matching each argument.

```c
int temperature;
double pressure;
scanf("%d %lf", &temperature, &pressure);
```

The `%d` placeholder tells `scanf` to read the sequence of characters as a decimal integer and convert it to the binary format of an `int`. The `%lf` tells `scanf` to read the next sequence of characters as a decimal real number and convert it to the binary format of a `double`. The binary values are then stored in the variables (memory locations) called temperature and pressure.

Note that each argument after the format string is prefixed with an ampersand (&). This is because in C, all arguments in function calls are passed by value, meaning that the function gets a copy of the value, rather than accessing the caller’s copy. (This is explained in detail in Chapter 24.) For example, consider the following `printf()` call:

```c
printf("The area is %.2f\n", area);
```
The printf() function only gets a copy of the value of area. It does not get direct access to the variable area. Hence, it is impossible for printf() to modify the value of area.

However, an input function like scanf() needs to modify variables in the calling function. We allow it to do so by passing it the memory address of each variable, rather than the value of the variable.

Recall that a variable is simply a name for a memory location where some value is stored. An ampersand (&) preceding a variable name represents the address of the variable, whereas the variable name alone would represent the value contained at that address.

By passing scanf() the address of a variable, we give it the ability to put something in that memory location.

Some of the most common placeholders for scanf() are outlined in Table 21.3.

<table>
<thead>
<tr>
<th>Type</th>
<th>Number Format</th>
<th>Placeholder</th>
</tr>
</thead>
<tbody>
<tr>
<td>char</td>
<td>printable character</td>
<td>%c</td>
</tr>
<tr>
<td>char</td>
<td>decimal</td>
<td>%hhd</td>
</tr>
<tr>
<td>short</td>
<td>decimal</td>
<td>%hd</td>
</tr>
<tr>
<td>int</td>
<td>decimal</td>
<td>%d</td>
</tr>
<tr>
<td>long int</td>
<td>decimal</td>
<td>%ld</td>
</tr>
<tr>
<td>long long int</td>
<td>decimal</td>
<td>%lld</td>
</tr>
<tr>
<td>float</td>
<td>decimal</td>
<td>%f</td>
</tr>
<tr>
<td>double</td>
<td>decimal</td>
<td>%lf</td>
</tr>
</tbody>
</table>

Table 21.3: Placeholders for printf

**Note** Note that unlike printf(), scanf() placeholders do not match multiple types. This is because the arguments to scanf() are addresses, not values, and are never promoted.

**Caution** Like gets(), scanf() may input a string too large for the character array provided, so it is considered a dangerous input function.

**Caution** Don’t use printf() or scanf() where a simpler function such as putchar() or puts() will do. Doing so is a waste of resources.

The printf() and scanf() functions do not directly support complex data types. Instead, it is left to the programmer to decide how complex numbers are represented in input and output as two separate float or double values. Functions such as creal() and cimag() can be used to extract the real and imaginary parts of a complex number for output in printf() as shown in Section 21.1.6.

You can see the full details of C I/O formatting on any Unix system by running *man 3 printf*. (Running *man printf* alone will display the man page for the printf command, not the printf function.)

### 21.1.6 Example of C input and output

```c
#include <stdio.h>
#include <sysexits.h>
#include <complex.h>

int main(int argc, char *argv[])
```
{
    double a, b, c;
    complex double root1, root2, two_a, discriminant_sqrt;

    printf("Please enter the coefficients a, b, and c: ");
    scanf("%lf %lf %lf", &a, &b, &c);
    
    /*
    * Precompute terms that will be used more than once.
    * Cast RESULT of all-double expression to complex so that only one
    * promotion occurs.
    * Convert 2a to complex now to avoid multiple promotions when
    * computing the roots.
    */
    discriminant_sqrt = csqrt((complex double)(b * b - 4.0 * a * c));
    two_a = 2.0 * a;

    root1 = (-b + discriminant_sqrt) / two_a;
    root2 = (-b - discriminant_sqrt) / two_a;

    printf("The roots of %fx^2 + %fx + %f are:
", a, b, c);
    printf("%g + %gi
", creal(root1), cimag(root1));
    printf("%g + %gi
", creal(root2), cimag(root2));
    return EX_OK;
}

Output from the program above:
Please enter the coefficients a, b, and c: 1 2 3
The roots of 1.000000x^2 + 2.000000x + 3.000000 are:
-1 + 1.41421i
-1 + -1.41421i

21.1.7 Using stderr with C

The puts(), and printf() functions are special cases that implicitly use stdout.

To print to stderr, we simply use the more general functions putc() and fputs(). These are the same functions we would use to print to any other file stream that our program opened, as discussed in Chapter 28.

fputs("Hello, world!", stdout); // Same as puts("Hello, world!");
fputs("Sorry, radius must be non-negative.\n", stderr);
fprintf(stderr, "Sorry, radius %f is invalid. It must be non-negative.\n", radius);

21.1.8 Self-test

1. What is a C stream? What are the three standard streams called in a C program?
2. What is the purpose of a C format string?
3. Write a C program that reads a line of input from the standard input using getc() or getchar() and echoes the characters to the standard output.
4. Show a C statements that prints "Hello, world!" to the standard output stream.
5. Show a C statements that prints "Hello, world!" to the standard error stream.
6. Does C share Fortran’s "one statement, one line" rule?
7. How to we make the C standard stream constants, such as `stderr`, available in our programs?

8. Write a C program that asks the user for the radius of a planet and prints the volume and surface area in clean sentence format as shown below.

   Be sure to write your numeric constants to match the data type of your variables.

   Please enter the radius of the planet in kilometers: 6371
   The area is 5.1e+08 km^2 and the volume is 1.1e+12 km^3.

9. Write a C program that asks the user for a value of x and prints the value of the following function:

   \[ y = x^5 - 5x^3 + \sin(3 - x^2) \]

   Verify the program output using a calculator to compute the function for several values of x.

21.2 Fortran Terminal Input/Output (I/O)

21.2.1 Write and Print

The general Fortran output statement is `write`. A write statement consists of the keyword `write` followed by `(unit, format)` and finally a list of variables and constants to be written, separated by commas.

\[
\text{write (unit, format) value [, value ...]}
\]

The unit number is an integer value that tells the write statement where to send the output. Each unit number represents a file stream associated with a file or output device such as a terminal screen, printer, plotter, etc. The file stream can be visualized as a conduit through which characters and numbers flow to or from I/O devices and files.

If a "*" is used in place of the unit number, the write statement writes to the standard output stream, which is normally a conduit to the terminal screen. (unless Unix redirection was used with the program).

The format string is a template for how the output should look. It specifies how many characters and how many fractional digits should be printed for real numbers, for example.

If a "*" is used in place of the format string, the write statement will use default formatting for all items printed (which is usually ugly, but functional).

After the (unit, format) come the items to be written. These may be variables of any type, string constants (characters between quotes), numeric constants, or algebraic expressions.

```fortran
real(8) :: height, width
write (*,*) 'Please enter height and width on one line:'
read (*,*) height, width
write (*,*) 'The area is ', height * width
```

The `print` statement is a short-hand for writing to the standard output.

\[
\text{print format, value [, value ...]}
\]

is equivalent to

\[
\text{write (*, format) value [, value ...]}
\]

Example:

```fortran
real(8) :: area
print *, 'The area is ', area
```
21.2.2 Read

The read statement inputs values from a stream such as the standard input and places the values into variables. The components of a read statement are the same as for a write statement.

Using a “*” in place of the unit number indicates that the read statement should read from the standard input stream, which is normally attached to the keyboard. Again, the standard input stream can be attached to other devices using redirection.

```fortran
real(8) :: height, width, area
write (*,*) 'Please enter the height and width on one line:'
read (*, *) height, width
area = height * width
write (*, *) 'The area is ', area
```

The read statement can also be written in short-hand form like the print statement:

```fortran
read *, height, width
```

21.2.3 One Statement, One Line

In Fortran, every read, write or print statement reads or writes a single line of input or output.

Hence, the read statement above expects to find both height and width on the same line of input. That is, the user should not press enter between the numbers.

Likewise, all of the output from a write or print statement will appear on the same line of output. The write or print statement will output all values, and send a CR/LF after the end of all output, so the next write or print will begin on a new line.

What is the output of the following?

```fortran
area = 4.0d0
write (*, *) 'The area is '
write (*, *) area
```

21.2.4 Standard Units

As mentioned earlier, using a “*” in place of the unit number tells the read statement to use the standard input stream and the write or print statement to use the standard output.

If you include the ISO_FORTRAN_ENV module in your program with:

```fortran
use ISO_FORTRAN_ENV
```

you can then use the actual unit numbers for the standard streams. The ISO_FORTRAN_ENV module provides the following named constants:

```
<table>
<thead>
<tr>
<th>Unit</th>
<th>Stream</th>
</tr>
</thead>
<tbody>
<tr>
<td>INPUT_UNIT</td>
<td>Standard Input</td>
</tr>
<tr>
<td>OUTPUT_UNIT</td>
<td>Standard Output</td>
</tr>
<tr>
<td>ERROR_UNIT</td>
<td>Standard Error</td>
</tr>
</tbody>
</table>
```

Table 21.4: Standard Stream Names

For the standard input and standard output, it’s easier to type “*” than INPUT_UNIT or OUTPUT_UNIT.

However, to write to the standard error unit, we must use ERROR_UNIT explicitly.
The standard error stream is, by default, attached to the terminal screen along with the standard output. However, programs should not regard them as equivalent.

Normal results from a program should be sent to the standard output stream.

Errors, warnings, and other information that is meant to inform the user about the condition of the program rather than present results from the computations, should be sent to the standard error.

Having two separate output streams for results and messages allows the user of the program to separate error and warning messages from normal output using redirection:

```
shell> a.out < input.txt > output.txt 2> errors.txt
```

This will be important to users who want to store screen output in a file for further processing. In this case, they will not want the output contaminated with error and warning messages.

Users may also want to see errors and warnings on the screen while redirecting results to a file. The following command reads keyboard input from the file `input.txt` and send results meant for the screen to `output.txt`. Anything the program prints to the standard error stream (ERROR_UNIT), however, will be written to the screen.

```
shell> a.out < input.txt > output.txt
```

### 21.2.5 Example of Fortran input and output

```fortran
!-----------------------------------------------------------------------
! Description:
! Find roots of a quadratic equation
! Usage:
! quadratic
! Returns:
! Nothing
!-----------------------------------------------------------------------

!-----------------------------------------------------------------------
! Modification history:
! Date   Name   Modification
! 2011-03-09 Jason Bacon Begin
!-----------------------------------------------------------------------

program quadratic
 ! Disable implicit declarations (i-n rule)
 implicit none

 ! Local variables
 real(8) :: a, b, c
 ! Use complex discriminant so we can get the square root 
 ! even if it’s negative.
 double complex :: discriminant_sqrt, root1, root2, two_a

 ! Input equation
 print *, 'Please enter the coefficients a, b, and c on one line:'
 read *, a, b, c

 ! Precompute terms that will be used more than once.
 ! Cast RESULT of all-double expression to complex so that only one
 ! promotion occurs.
 ! Convert 2a to complex now to avoid multiple promotions when
 ! computing the roots.
 discriminant_sqrt = sqrt(dcmplx(b * b - 4.0d0 * a * c))
```

two_a = 2.0d0 * a

! Compute roots
root1 = (-b + discriminant_sqrt) / two_a
root2 = (-b - discriminant_sqrt) / two_a

! Output roots
print *, 'The roots of ', a, 'x^2 + ', b, 'x + ', c, ' are:
print *, root1
print *, root2
end program

Output from the program above:
Please enter the coefficients a, b, and c on one line:
1 2 3
The roots of 1.0000000000000000 x^2 + 2.0000000000000000 x + 3.0000000000000000 are:
( -1.0000000000000000 , 1.4142135623730951 )
( -1.0000000000000000 , -1.4142135623730951 )

21.2.6 Fortran Formatted I/O

As mentioned, one of the asterisks in the write, print, and read statements tells the statement to use default formatting.
The default format is designed to use a consistent number of columns and present accurate information. This often leads to
excessive white space and decimal places in the output:

integer :: i = 5
real(8) :: x = 5.0d0

! Statements
print *, i, x

Output: 5 5.0000000000000000
Ruler: 123456789012345678901234567890123

As you can see from the "ruler" below the output, the integer is printed across 12 columns, and the real(8) value across 21.
If we want to control the appearance of output or validate the format of input, we can replace the asterisk with a format specifier.
The format specifier contains descriptors for each item to be printed, in the same order as the arguments to the read, write or
print statement. The first descriptor describes the format of the first argument, and so on.
The most common descriptors are described in Table 21.5.

<table>
<thead>
<tr>
<th>Descriptor</th>
<th>Type</th>
<th>Format</th>
</tr>
</thead>
<tbody>
<tr>
<td>Iw</td>
<td>Any integer type</td>
<td>w columns total</td>
</tr>
<tr>
<td>Fw.d</td>
<td>Any real type</td>
<td>w columns total, d decimal places</td>
</tr>
<tr>
<td>Ew.d</td>
<td>Any real type</td>
<td>w columns total, d decimal places, scientific notation</td>
</tr>
<tr>
<td>Aw</td>
<td>String</td>
<td>w columns total</td>
</tr>
<tr>
<td>Gw.d</td>
<td>Any real type</td>
<td>Program chooses the best format</td>
</tr>
</tbody>
</table>

Table 21.5: Common Format Descriptors

integer :: i = 5, j = -6
real(8) :: x = 5.0d0, y = -6.0d0
! Statements
print '(i3, f10.2)', i, x
Output: 5 5.00
Ruler: 1234567890123

write (*, '(i3, f10.2)'), i, x
Output: 5 5.00
Ruler: 1234567890123

print '(i3, e10.2)', i, x
Output: 5 0.50E+01
Ruler: 1234567890123

print '(i3, g10.2)', i, x
Output: 5 5.0
Ruler: 1234567890123

print '(i3, i3, f10.2, f10.2)', i, j, x, y
Output: 5 -6 5.00 -6.00
Ruler: 12345678901234567890123456

If the width is non-zero and the value to be printed doesn’t fit, the program will print a ‘*’ instead of the number:

integer :: i = 5, j = -6
real(8) :: x = 5.0d0, y = -6.0d0

! Statements
print '(i1, i1, f4.2, f4.2)', i, j, x, y
Output: 5*5.00****
Ruler: 1234567890

If the width is 0, the program will print the minimum number of digits needed to convey the correct value:

integer :: i = 5, j = -6
real(8) :: x = 5.0d0, y = -6.0d0

! Statements
print '(i0, i0, f0.2, f0.2)', i, j, x, y
Output: 5-65.00-6.00
Ruler: 123456789012

The width and decimal places specifiers can be omitted for string values, in which case the program will print the string to its full length. This feature can be used with the zero-width descriptor for embedding numbers in text:

integer :: i = 5, j = -6
real(8) :: x = 5.0d0, y = -6.0d0

! Statements
print '(a, i0, a, i0, a, f0.2, a, f0.2)', &
i = ’’, i = ’’, j = ’’, j = ’’, x = ’’, x = ’’, y = ’’, y
Output: i = 5 j = -6 x = 5.00 y = -6.00
Ruler: 12345678901234567890123456
A format descriptor, or group of format descriptors in () can be preceded by an integer repeat count to shorten the format specifier:

```fortran
print '(i3, i3, f10.2, f10.2)', i, j, x, y
```

same as

```fortran
print '(2i3, 2f10.2)', i, j, x, y
```

```fortran
print '(a, i0, a, i0, a, f0.2, a, f0.2)', &
     'i = ', i, ' j = ', j, ' x = ', x, ' y = ', y
```

same as

```fortran
print '(2(a, i0), 2(a, f0.2))', &
     'i = ', i, ' j = ', j, ' x = ', x, ' y = ', y
```

Format specifiers can also be used in read statements. If a format specifier is used, then the input must match the format exactly, or an error will be generated. This is done to perform strict checking on input read from a file, if slight deviations in the format might indicate a problem with the data.

If we want to use the same format specifier in multiple print, read, or write statements, we can separate it out to a labeled format statement, and use the label in its place:

```fortran
print 10, 'i = ', i, ' j = ', j, ' x = ', x, ' y = ', y
print 10, 'k = ', k, ' l = ', l, ' v = ', v, ' w = ', w
10 format (2(a, i0), 2(a, f0.2))
```

### 21.2.7 Self-test

1. What is a Fortran unit number?
2. What is the purpose of a Fortran format string?
3. Show a Fortran statement that prints "Hello, world!" to the standard output stream.
4. Show a Fortran statement that prints "Hello, world!" to the standard error stream.
5. Explain the Fortran "one statement, one line" rule.
6. How to we make the Fortran standard unit constants, such as ERROR_UNIT, available in our programs?
7. Write a Fortran program that asks the user for the radius of a planet and prints the volume and surface area in clean sentence format as shown below. Be sure to write your numeric constants to match the data type of your variables.

   ```fortran
   Please enter the radius of the planet in kilometers: 6371
   The area is .51E+009 km^2 and the volume is .11E+013 km^3.
   ```

8. Write a Fortran program that asks the user for a value of \( x \) and prints the value of the following function:

   \[
   y = x^5 - 5x^3 + \sin(3-x^2)
   \]

   Verify the program output using a calculator to compute the function for several values of \( x \).
Chapter 22

Conditional Execution

22.1 Motivation

Computers are much more useful when they can make decisions. Most calculations cannot proceed very far without reaching some sort of crossroad, where a decision must be made about what to do next.

22.2 Design vs. Implementation

Most decisions involved in a process should be discovered during the design phase. The implementation phase then remains primarily a process of translation.

Top-down design

22.3 Boolean Expressions

Decisions are based on Boolean expressions, which have a value of true or false.

The term Boolean is named after George Boole, a well-known mathematician, philosopher, and logician, who developed a systematic approach to logic now known as Boolean algebra. The basis of Boolean algebra was published in his text "The Laws of Thought", and provides a significant part of the foundation of computer science.

Fortran supports decisions using the logical data type.

The C language does not have a separate Boolean data type. Recall that the design philosophy of C is minimalist and explicit rather than bloated and abstract.

Boolean values in C are represented as they are stored internally, as integer values. A value of 0 is interpreted as false and any non-zero value is interpreted as true. A derived Boolean type called bool is provided in the standard header file stdbool.h.

22.3.1 Boolean Constants

Fortran provides logical constants .true. and .false.. The periods are a vestige of early Fortran, which used periods to delineate many program elements (tokens) in order to make parsing easier.
Since the C language does not have a separate Boolean data type, there are no built-in constants. However, the values `true` and `false` are defined in `<stdbool.h>`.

```c
#include <stdbool.h>
#include <sysexits.h>

int main()
{
  bool raining;
  raining = false;
  ...
  return EX_OK;
}
```

### 22.3.2 Relational Expressions

A relational expression is a Boolean expression that compares (tests the relation between) two values of any type, such as integers, real numbers, or strings. The relational expression itself has a value of either true or false, while the values being compared could be of any type.

<table>
<thead>
<tr>
<th>Expression</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>1 &gt; 2</code></td>
<td>false</td>
</tr>
<tr>
<td><code>5 = 10/2</code></td>
<td>true</td>
</tr>
<tr>
<td>&quot;Bart&quot; = &quot;Bert&quot;</td>
<td>false</td>
</tr>
</tbody>
</table>

Table 22.1: Relational Expressions

Relational expressions in Fortran always have a value of `.false.` or `.true.`.

Relational expressions in C have a value of 0 or 1, which can be represented as false and true if `stdbool.h` is included in the program.

### Relational Operators

Relational expressions are formed using Relational operators. Fortran and C provide the usual arithmetic relations such as "equals", "less than", etc. Fortran 90 and later use more intuitive symbols for operators, while Fortran 77 and earlier use old style operators.

Fortran 90 is backward-compatible with Fortran 77, so the old style operators can still be used. This allows old Fortran 77 code to be compiled with the newer compilers, but newly written code should use the new operators.

<table>
<thead>
<tr>
<th>Relational Operator</th>
<th>Fortran 77 and Earlier</th>
<th>Fortran 90 and Later</th>
<th>C</th>
</tr>
</thead>
</table>
| Not equal           | .ne.                   | /=                   | !=
| Equals              | =eq.                   | ==                   | == |
| Less than           | .lt.                   | <                    | <  |
| Less or equal       | .le.                   | <=                   | <= |
| Greater than        | .gt.                   | >                    | >  |
| Greater or equal    | .ge.                   | >=                   | >= |

Table 22.2: Fortran Relational Operators

Fortran 90 adopted the operators from C to replace the arcane .ne., etc., except for !=, which would mark the beginning of a comment in Fortran.
Caution
Floating point values should NEVER be compared for equality. Due to round-off error, it is possible that the actual value might be slightly different than the expected value and therefore not equal to the value we’re looking for. For example, the following condition is likely to fail, because 0.1 cannot be accurately represented in binary floating point. Even if it could be represented accurately, the value of thickness might be off due to round off during computations.

```fortran
if ( thickness == 0.1 )
```

### Boolean Operators

Some Boolean expressions entail more than one relation. For example, an airline may want to offer free peanuts to both kids and seniors, in which case we would want to know that either age < 18 or age >= 65.

<table>
<thead>
<tr>
<th>Operation</th>
<th>Fortran Operator</th>
<th>C Operator</th>
</tr>
</thead>
<tbody>
<tr>
<td>and</td>
<td>.and.</td>
<td>&amp;&amp;</td>
</tr>
<tr>
<td>or</td>
<td>.or.</td>
<td></td>
</tr>
<tr>
<td>not</td>
<td>.not.</td>
<td>!</td>
</tr>
</tbody>
</table>

Table 22.3: Boolean Operators

#### 22.3.3 Self-test

1. What data type does Fortran use for Boolean expressions?
2. What data type does C use for Boolean expressions?

#### 22.4 Flow Charts

TBD

#### 22.5 If Statements

The Fortran `if-then-else-endif` statement is used to test Boolean expressions and conditionally execute sections of code.

##### 22.5.1 Minimal If Statements

The minimal version of conditional execution consists of a condition and a single statement.

```fortran
if ( condition ) statement
```

```fortran
if ( condition ) &
  statement
```

*Note* In Fortran, the statement must be on the same line as the `if ( condition )`, unless a continuation character `&` is used. If the statement is below the `if`, it should be indented one more level than the `if`. 
A simple if statement looks virtually identical in C. C is free-format, so no continuation character is needed to span multiple lines and the statement is usually placed on the line below, indented one more level than the if ( condition ).

```c
if ( (age < 18) || (age >= 65) )
    puts("You get free peanuts.");
```

### 22.5.2 If with Multiple Statements

If more than one statement are to be conditionally executed, then Fortran requires the longer form, with the word "then" on the same line as the if, the statements on subsequent lines, and finally "endif" on a line below the last statement. The statements between the if and endif are indented one more level than the if and endif.

```fortran
if ( (age < 18) .or. (age >= 65) ) then
    print *, 'You get free peanuts.'
    peanut_inventory = peanut_inventory - 1
endif
```

In C, including multiple statements under an if or any other construct uses the same syntax. A C block statement is any group of statements enclosed in curly braces ({ }). Block statements can be used anywhere in a C program, such as in if statements, loops, or even alone.

```c
// A pointless, but valid use of {}
{ puts("You get free peanuts!"); }
```

Since C is a free-format language, it makes no difference to the compiler whether the curly braces are on the same line or different lines. However, indentation is important for readability. Typically, the braces are aligned in the same column and the statements between the braces are indented one more level.

```c
if ( (age < 18) || (age >= 65) )
{
    puts("You get free peanuts!");
    peanut_inventory = peanut_inventory - 1;
}
```

### 22.5.3 Else

The basic if statement above executes one or more statements if some condition is true, and does nothing otherwise. Sometimes, we want to do two different things depending whether a condition is true or false.

```fortran
if ( (age < 18) .or. (age >= 65) ) then
    print *, 'You get free peanuts.'
    peanut_inventory = peanut_inventory - 1
else
```
print *, 'No free peanuts for you!'
endif

if ( (age < 18) || (age >= 65) )
{
    puts("You get free peanuts!");
    peanut_inventory = peanut_inventory - 1;
} else
    puts("No free peanuts for you!");

In some cases, we might be tempted to use an if statement to set a logical variable for later use:

if ( (age < 18) .or. (age >= 65) ) then
    free_peanuts = .true.
else
    free_peanuts = .false.
endif

if ( (age < 18) || (age >= 65) )
    free_peanuts = true;
else
    free_peanuts = false;

However, note that we are simply setting free_peanuts to the same value as the condition in the if statement. In cases like this, we can replace the entire if-else statement with a simple assignment:

free_peanuts = (age < 18) .or. (age > 65)
free_peanuts = (age < 18) || (age > 65);

### 22.5.4 If-then-elseif-...-elseif-else-endif

We can take it a step further, and check for a series of conditions:

if ( age < 3 ) then
    print *, 'No peanuts for you! You might choke on them.'
else if ( age < 10 ) then
    print *, 'Ask your mom if you can have some peanuts.'
else if ( (age < 18) .or. (age >= 65) ) then
    print *, 'You get free peanuts!'
    peanut_inventory = peanut_inventory - 1
else
    print *, 'No free peanuts for you!'
endif

if ( age < 3 )
    puts("No peanuts for you! You might choke on them.");
else if ( age < 10 )
    puts("Ask your mom if you can have some peanuts.");
else if ( (age < 18) || (age >= 65) )
{
puts("You get free peanuts!");
    peanut_inventory = peanut_inventory - 1;
} else
    puts("No free peanuts for you!");

22.5.5 Nesting If Statements

The code inside an if or else clause can contain anything, including other if statements. Suppose the economy is tight, and we
decide to cancel free peanuts for seniors, but keep them for kids. We might then restructure the code as follows:

```c
if ( age < 18 ) then
    if ( age < 3 ) then
        print *, 'No peanuts for you! You might choke on them.'
    else if ( age < 10 ) then
        print *, 'Ask your mom if you can have some peanuts.'
    else
        print *, 'You get free peanuts!'
        peanut_inventory = peanut_inventory - 1
    endif
else
    print *, 'No free peanuts for you!'
endif
```

```c
if ( age < 18 )
    if ( age < 3 )
        puts("No peanuts for you! You might choke on them.");
    else if ( age < 10 )
        puts("Ask your mom if you can have some peanuts.");
    else
    
    puts("You get free peanuts!");
    peanut_inventory = peanut_inventory - 1;
    
else
    puts("No free peanuts for you!");
```

22.5.6 Self-test

1. Write a C or Fortran program that asks the user for the coefficients A, B, and C or a quadratic equation. If the discriminant
is non-negative, the program should print the real roots to the standard output. Otherwise, it should print an error message
to the standard error.

```
Please enter the coefficients a, b, and c: 1 2 1
The roots of 1.000000x^2 + 2.000000x + 3.000000 are:
-1
-1
Please enter the coefficients a, b, and c: 1 2 3
Error: This equation has no real roots.
```

2. Nested if: TBD
22.6 Switch, Select-Case

If we need to compare a single expression to a number of different constant values, we can use the Fortran case-select statement or the C switch statement. The value being compared is called the selector value.

```fortran
select case (age)
  case (0:2)
    print *, 'No peanuts for you! You might choke on them.'
  case (3:9)
    print *, 'Ask your mom if you can have some peanuts.'
  case (10:17)
    print *, 'You get free peanuts!'
    peanut_inventory = peanut_inventory - 1
  case default
    print *, 'No free peanuts for you!'
end case
```

```c
switch(age)
{
  case 0:
  case 1:
  case 2:
    puts("No peanuts for you! You might choke on them.");
    break;
  case 3:
  case 4:
  case 5:
  case 6:
  case 7:
  case 8:
  case 9:
    puts("Ask your mom if you can have some peanuts.");
    break;
  case 10:
  case 11:
  case 12:
  case 13:
  case 14:
  case 15:
  case 16:
  case 17:
    puts("You get free peanuts!");
    peanut_inventory = peanut_inventory - 1;
    break;
  default:
    puts("No free peanuts for you!");
}
```
Note

In the Fortran case-select statement, control jumps to the end automatically after executing the chosen case. In the C switch statement, this is not true. The switch statement causes a jump to the correct case, but execution will continue into the cases that follow until a break statement is encountered. This feature is occasionally useful where the statements that should be executed for case ‘B’ are a subset of those for case ‘A’. In this situation, we simply place case ‘B’ after case ‘A’ and omit the break after case ‘A’.

```
switch(val)
{
    case CHEESE:
        puts("This selection is high in fat.");
        // Continue into case B below, since cheese is high
        // in calories as well.
        case POTATOES:
        case PRETZELS:
            puts("This selection is high in calories.");
            break;
}
```

Note The C switch statement does not accept ranges of values as Fortran does. However, the flow-through feature described above allows us to put multiple cases back-to-back (with no code between them) to effectively construct a single case for multiple values. This is another example of C’s minimalist philosophy, avoiding the addition of another feature that would not add functionality, but only make the code prettier.

The selector value must be an integral type, i.e. integer, logical, or character. Real numbers are a continuous type, and floating point’s limited precision makes comparison for equality uncertain.

A switch or select-case statement is sometimes more efficient than the equivalent if-then-elseif...endif series. The if-then-elseif...endif has to check every value until it finds the one that matches. Given 40 case values, and a uniform probability that the selector value will match any one of them, the if-then-elseif...endif series will average 20 comparisons.

A switch of select-case statement, on the other hand, is often compiled to machine code that uses a jump table, which is a list of addresses for each case. By converting the selector value to a position in the jump table, the select-case can jump directly to any case without comparing the selector value to any other cases. This is about a 20-fold speed-up when there are 40 cases.

### 22.6.1 Self-test

1. Write a C or Fortran program that implements a simple menu-driven calculator program.

```
1.. Sine
2.. Cosine
3.. Tangent
9.. Quit

1
What is the angle in degrees? 45
The sine is 0.707106781187.
```

### 22.7 Code Quality

Indentation
22.8 Performance

60% of if conditions are false

```c
if ( a == b )
  c = 1;
else
  c = 2;
d = 5;
```

# The if clause below requires an added "b done" instruction to exit the block
# The else clause is therefore faster.

```assembly
cmp a, b
bne else

# If clause (a == b)
    mov c, 1
    b done

# Else clause (a != b)
else mov c, 2

# First instruction after if block
done mov d, 5
```

Don’t do silly things

22.8.1 Code Examples

```c
if ( a == b )
  equals = true;

// Faster
equals = (a == b)
```
Chapter 23

Loops (Iterative Execution)

23.1 Motivation

Computers are good at performing large numbers of calculations fast. Suppose solving a problem requires performing the same calculations on a billion different inputs. One way we could achieve this is by writing a billion input statements, each followed by a Fortran statement which works on the latest input. Obviously, this wouldn’t be very productive.

Loops, also known as iteration, provide a way to use the same statements repeatedly for different data.

23.2 Design vs. Implementation

The iterative nature of a solution should always come out in the design stage of development, not during implementation. Regardless of the complexity of a problem, implementation should always be a fairly simple process of translating the design.

23.3 Anatomy of a Loop

A loop consists of a several parts:

- Initialization: Code that sets initial values before the first iteration.
- Condition: A Boolean expression that determines whether the loop will iterate again. The condition is checked before or after each iteration.
- Body: The statements inside the loop, which are executed once during each iteration.
- Housekeeping: Overhead for controlling the loop, such as incrementing a counter variable.

23.3.1 Self-test

1. Describe the anatomy of a loop.

23.4 While: The Universal Loop

The while loop is the only loop that needs to exist. All other loops exist for the sake of convenience.

The basic structure of a Fortran while loop looks like this:
initialization
do while ( condition )
  body
  housekeeping
enddo

A C while loop looks similar:

initialization
while ( condition )
  body and housekeeping

initialization
while ( condition )
{
  body
  housekeeping
}

The condition can be any Boolean expression, i.e. any expression with a value of .true. or .false. in Fortran, and any integer expression in C.

Example 23.1 Do-while Loop
Suppose we want to know the sine of every angle from 0 to 359 degrees. We could write 360 print statements, or we could use the following loop:

```fortran
!-----------------------------------------------------------------------
! Program description:
! Print sine of every angle in degrees from 0 to 360
!-----------------------------------------------------------------------
module constants
double precision, parameter :: PI = 3.1415926535897932d0
end module constants

program sine_loop
  use constants ! Constants defined above
  implicit none
  real(8) :: angle

  ! Initialization
  angle = 0.0d0

  ! Condition
  do while ( angle <= 360.0d0 )
    ! Body
    print *, 'sine(', angle, ') = ', sin(angle * PI / 180.0d0)
    ! Housekeeping
    angle = angle + 1.0d0
  end do
end program
```

```c
/***************************************************************************
* Description:
* Print sine of every angle in degrees from 0 to 360
***************************************************************************/
```
#include <stdio.h>
#include <math.h>
#include <sysexits.h>

int main(int argc,char *argv[])
{
    double angle;

    // Initialization
    angle = 0.0;

    // Condition
    while ( angle <= 360.0 )
    {
        // Body
        printf("sine(%f) = %f\n", angle, sin(angle * M_PI / 180.0));

        // Housekeeping
        angle += 1.0;
    }
    return EX_OK;
}

Since C Boolean expressions are integers, it is possible to do some odd-looking things in a C program. For instance, since false is 0, you could loop down from some number to 0 without using a relational operator:

c = 10;
while ( c )
{
    printf("%d squared = %d\n", c, c * c);
    --c;
}

Some programmers will do this thinking that it makes them look clever, but it’s really just cryptic, and not a good practice. It’s better to be explicit and clear, even if it just saves the reader a few seconds of thinking.

c = 10;
while ( c > 0 )
{
    printf("%d squared = %d\n", c, c * c);
    --c;
}

Example 23.2 A More Flexible Sine Printer

!-----------------------------------------------------------------------
! Program description:
! Print sine of every angle in degrees from 0 to 360
!-----------------------------------------------------------------------

module constants
    double precision, parameter :: PI = 3.1415926535897932d0
end module constants

program sine_loop
    use constants ! Constants defined above

    implicit none
    real(8) :: angle, final_angle
The condition in a do-while loop can be literally any Boolean expression. Programmers can use their imagination and construct loops with any conceivable condition. The examples above barely scratch the surface of what is possible with do-while loops.

### 23.4.1 Self-test

1. Write a C or Fortran program that prints the square root of every integer from 0 to 10.

```
sqrt(0) = 0.000000
sqrt(1) = 1.000000
sqrt(2) = 1.414214
sqrt(3) = 1.732051
```
23.5 Fortran Fixed Do Loops

Fortran provides another form of loop for convenience, since iterating over a fixed set of values is so common. The basic structure is as follows:

```fortran
do variable = start-value, end-value, stride
  body
enddo
```

The stride value is automatically added to the loop variable after the body is executed. This loop is equivalent to

```fortran
variable = start-value
do while ( variable <= end-value )
  body
  variable = variable + stride
enddo
```

This loop combines the initialization, condition, and housekeeping into one line. It does not allow complex Boolean conditions: it only works for loops that go from a starting value to an ending value. All of the values can be variables.

One limitation of this loop is that the loop variable must be an integer.

The stride is optional, and if omitted, defaults to 1, regardless of the start and end values.

**Example 23.3 Predicting Loop Output**

What is the output of each of the following loops?

```fortran
integer :: angle
!
! Initialization, condition, and housekeeping
do angle = 0, 359, 1
  ! Body
  print *, 'sine(',angle,') = ', sin(angle * PI / 180.0d0)
enddo

integer :: angle
!
! Initialization, condition, and housekeeping
do angle = 0, 359
  ! Body
  print *, 'sine(',angle,') = ', sin(angle * PI / 180.0d0)
enddo

integer :: angle
!
! Initialization, condition, and housekeeping
do angle = 359, 0
  ! Body
  print *, 'sine(',angle,') = ', sin(angle * PI / 180.0d0)
enddo
```
23.5.1 Self-test

1. Write a Fortran program that prints the square root of every integer from 0 to 10 using a fixed do loop.

   ```fortran
   sqrt(0) = 0.000000
   sqrt(1) = 1.000000
   sqrt(2) = 1.414214
   sqrt(3) = 1.732051
   sqrt(4) = 2.000000
   sqrt(5) = 2.236068
   sqrt(6) = 2.449489
   sqrt(7) = 2.645751
   sqrt(8) = 2.828427
   sqrt(9) = 3.000000
   sqrt(10) = 3.162278
   ```

23.6 The C for Loop

C also has a convenience loop that combines the initialization, condition, and housekeeping into the beginning of the loop, but unlike Fortran's do loop, it is not less flexible than a while loop.

The C for loop is a condensed while loop. It simply collects the initialization, condition, and housekeeping into one place for the sake of readability.

```c
for (initialization; condition; housekeeping)
    body
```

```c
/***************************************************************************
* Description:
* Print sine of every angle in degrees from 0 to 360
***************************************************************************/
#include <stdio.h>
#include <math.h>
#include <sysexits.h>

int main(int argc,char *argv[])
{
    double angle;
    // Condition
    for ( angle = 0.0; angle <= 360.0; angle += 1.0 )
        // Body
            printf("sine(%f) = %f\n", angle, sin(angle * M_PI / 180.0));
    return EX_OK;
}
```

23.6.1 Self-test

1. Write a C program that prints the square root of every integer from 0 to 10 using a for loop.

   ```c
   sqrt(0) = 0.000000
   sqrt(1) = 1.000000
   sqrt(2) = 1.414214
   sqrt(3) = 1.732051
   sqrt(4) = 2.000000
   sqrt(5) = 2.236068
   ```
23.7 Fortran Unstructured Do Loops

An unstructured do loop does not have a built-in condition check, but uses an if statement to exit the loop when some condition is met.

Example 23.4 Unstructured Loop

```fortran
! Initialization
angle = 0.0d0
do
  ! Body
  print *, 'sine(',angle,') = ', sin(angle * PI / 180.0d0)
  ! Housekeeping and condition
  angle = angle + 1.0d0
  if ( angle > 359.0d0 ) exit
enddo
```

The advantage of an unstructured do loop is that the condition can be checked anywhere within the body of the loop. Structured loops always check the condition before the body is executed. The down side to unstructured do loops is that they are a bit harder to read, so they should be avoided if possible.

23.7.1 Self-test

1. Write a Fortran program that prints the sine of every angle entered by the user until they enter a sentinel value of 361. Note: the angle should be input as an integer in degrees so that it can be compared to the sentinel value using `==`.

```
Please enter an integer angle in degrees, or 361 to quit:
4
The sine of 4 is .07.
Please enter an integer angle in degrees, or 361 to quit:
20
The sine of 20 is .34.
Please enter an integer angle in degrees, or 361 to quit:
100
The sine of 100 is .98.
Please enter an integer angle in degrees, or 361 to quit:
-12
The sine of -12 is -.21.
Please enter an integer angle in degrees, or 361 to quit:
361
```

23.8 The C break Statement

Like Fortran's `exit` statement, the C `break` statement immediately terminates the current loop so the program continues from the first statement after the loop.

Unlike Fortran, C does not have a unconditional loop construct, so use of `break` is rarely useful. Well-structured code will generally use the loop condition to terminate the loop, and the if statement required to trigger a break would be redundant.
23.8.1 Self-test

1. TBD

23.9 The C do-while Loop

There are situations where we want a loop to iterate at least once. This is especially common where a loop performs input and terminates on some sort of *cardinal value*, a special value used to mark the end of input. Using a while or for loop, this requires setting values before the loop begins to ensure that the condition is true. This is known as *priming* the loop. The term comes from the old practice of manually spraying some fuel into the carburetor of a cold engine in order to help it start.

```c
age = 0;
while ( age != -1 )
{
    printf("Enter an age. Enter -1 when done: ");
    scanf("%d\n", &age);
    ...
}
```

C offers another type of loop that checks the condition *after* the body is executed, so that priming the loop is unnecessary.

```c
do
{
    printf("Enter an age. Enter -1 when done: ");
    scanf("%d\n", &age);
    ...
} while ( age != -1 );
```

23.9.1 Self-test

1. Write a C program that prints the sine of every angle entered by the user until they enter a sentinel value of 361. Note: the angle should be input as an integer in degrees so that it can be compared to the sentinel value using ==.

```
Please enter an integer angle in degrees, or 361 to quit:
4
The sine of 4 is .07.
Please enter an integer angle in degrees, or 361 to quit:
20
The sine of 20 is .34.
Please enter an integer angle in degrees, or 361 to quit:
100
The sine of 100 is .98.
Please enter an integer angle in degrees, or 361 to quit:
-12
The sine of -12 is -.21.
Please enter an integer angle in degrees, or 361 to quit:
361
```

23.10 Fortran cycle and C continue

The *cycle* and *continue* statements causes the current iteration of a loop to end without executing the rest of the body. Rather than terminating the loop, as *exit* does, it then goes back to the beginning of the loop and possibly begins the next iteration. Use of *cycle* and *continue* is unstructured, and makes code difficult to read, so its use should be avoided.
23.11 “Infinite” Loops

It is possible with most types of loops to create a condition that will always be true. Some conditions are simply always true, and others may be always true for a particular set of inputs.

Such a condition causes what is called an infinite loop. A program with an infinite loop will continue to execute indefinitely, until it is killed (e.g. using Ctrl+C or the kill command in Unix).

A common way to create an infinite loop is by forgetting to add the housekeeping code to a do-while loop.

Example 23.5 Infinite Loop

When constructing a loop, we must take care to ensure that the condition will eventually become false. Otherwise, the loop will simply continue to run until the program is killed by the user, a power outage, or some other external event. This is known as an infinite loop.

The loops below fail to update angle, and therefore continue to print sine(0) indefinitely.

```
! Initialization
angle = 0.0d0
final_angle = 360.0d0

! Condition
do while ( angle <= final_angle )
  ! Body
  print *, 'sine(',angle,') = ', sin(angle * PI / 180.0d0)
endo

! Initialization
angle = 0.0
final_angle = 360.0

! Condition
while ( angle <= final_angle )
{
  printf("sine(%f) = %f\n", angle, sin(angle * M_PI / 180.0));
}
```

Sometimes it’s obvious that an infinitely loop is occurring, because of the output the program is producing. Other times, if the program is producing output during the loop, it may be harder to detect.

23.11.1 Self-test

1. What is a common cause of infinite loops?
2. How do we terminate a process that’s stuck in an infinite loop?

23.12 Loops and Round-off Error

Floating point limitations can wreak havoc with loops if you’re not careful. What would you expect to be the output of the following loop?
! Main program body
program zero2one
    implicit none
    real(8) :: x

    x = 0.0d0
    do while ( x /= 1.0d0 )
        print *, x
        x = x + 0.1d0
    enddo
end program

#include <stdio.h>
#include <sysexits.h>

int main(int argc, char *argv[])
{
    double x;

    x = 0.0;
    while ( x != 1.0 )
    {
        printf("%0.16f
", x);
        x += 0.1;
    }
    return EX_OK;
}

It would seem reasonable to expect that it prints the numbers from 0.0 through 0.9 in increments of 0.1.

0.0000000000000000
0.1000000000000000
0.2000000000000000
0.3000000000000000
0.4000000000000000
0.5000000000000000
0.6000000000000000
0.7000000000000000
0.8000000000000000
0.9000000000000000

The actual results are shown below:

0.0000000000000000
0.1000000014901161
0.2000000029802323
0.3000000044703484
0.4000000059604645
0.5000000074505806
0.6000000089406967
0.7000000104308128
0.8000000119209289
0.9000000134110451
1.000000014901161
1.100000016391277
1.200000017881393
1.300000019371509
1.400000020861625
What causes this problem?

How can it be avoided?

In this case, the problem is caused by the fact that 0.1\textsubscript{10} cannot be represented in binary. It’s much like trying to represent 1/3 in decimal: it requires an infinite number of digits.

The ==, !=, and /= operators should never be used with floating point. Because floating point values are prone to round-off error, floating point comparisons must always include a tolerance. We might be tempted to think the code below avoids the problem, but it does not.

```fortran
real(8) :: x
x = 0.0d0
do while ( x < 1.0d0 )
  print *, x
  x = x + 0.1
enddo
```

The problem with this code is that different floating point hardware may produce different round off errors. For example, if round-off results in a value of 0.999999999788713 instead of 1.0000000149011612, then this loop will perform one extra iteration, ending at approximately 1.0 instead of approximately 0.9 as we intended.

The real solution is to use a tolerance. In this case, we want to stop when x is about 1.0. We should always choose the largest tolerance that will work, in case round-off accumulates to a high level. Using half the increment will work for this loop. It will stop the loop when x is between 0.95 and 1.05.

```fortran
real(8) :: x, increment, tolerance
increment = 0.1
tolerance = increment / 2.0d0
x = 0.0d0
do while ( abs(x-1.0) > tolerance )
  print *, x
  x = x + increment
enddo
```

Another possible solution is to use a separate integer loop variable:

```fortran
real(8) :: x, increment, tolerance
integer :: counter
increment = 0.1
tolerance = increment / 2.0d0
x = 0.0d0
counter = 0
do while ( counter /= 10 )
  print *, x
  x = x + increment
  counter = counter + 1
enddo
```
Both of these solutions are somewhat unsatisfactory, since they require extra code to solve the problem. However, this is sometimes the nature of programming, especially when dealing with an imperfect system such as floating point.

23.12 Self-test

1. What types of problems can round-off error cause with loops?

2. How can we avoid these kinds of problems?

23.13 Examples

Loops can be used in many ways. They can perform the same calculations on many different starting values, as shown in the sine() examples above.

Loops are also commonly used to perform a series of calculations on a single starting value, to produce a single result.

A common use for this sort of iteration is found in the field of numerical analysis, which uses iterative calculations to solve many kinds of equations.

23.13.1 Integer Powers

Integer powers can be computed using a simple loop.

```fortran
!-----------------------------------------------------------------------
! Program description:
!       Compute integer powers of real numbers
!-----------------------------------------------------------------------

!-----------------------------------------------------------------------
! Modification history:
! Date       Name       Modification
! 2011-03-10  Jason Bacon   Begin
!-----------------------------------------------------------------------

! Main program body
program int_power
  use constants ! Constants defined above
  use ISO_FORTRAN_ENV ! INPUT_UNIT, OUTPUT_UNIT, ERROR_UNIT, etc.

  ! Disable implicit declarations (i-n rule)
  implicit none

  ! Variable definitions
  real(8) :: base, power
  integer :: exponent, i

  ! Statements
  print *, 'Enter base and exponent on one line:
  read *, base, exponent
  power = 1.0d0
  do i = 1, exponent
     power = power * base
  enddo
  print *, base, '**', exponent, ' = ', power
end program
```
# Variable definitions

double base, power;
unsigned int exponent, i;

// Statements
printf("Enter base and exponent: ");
scanf("%lf %u", &base, &exponent);
power = 1.0;
for (i = 1; i <= exponent; ++i)
   power *= base;
printf("%f ** %u = %f\n", base, exponent, power);
return EX_OK;

## 23.13.2 Newton's Method

A simple example is Newton’s method for finding the roots of an equation.

![Newton's Method Diagram](image-url)
1. Specification: Find a root for a function $f(x)$. (Where does $f(x)$ equal 0?)

2. Design: Use Newton's method:
   
   (a) Make an initial guess for $x$.
   
   (b) Follow a tangent line from the function $f(x)$ to where it crosses the x axis. This point will usually be closer to the root of $f(x)$ than the initial $x$ was. The value of $x$ where the tangent intersects the axis is: $n = x - f(x) / f'(x)$
   
   (c) Use $n$ as the next guess. By repeating the process, we get closer to the root of $f(x)$ each time.

How do we know when to stop iterating?

- Method 1: Check the value of $f(x)$. The problem with this method is that a function could come very close to zero at some point, and then veer away, so we could get a false positive root detection.

- Method 2: Compare guesses. As we get closer to a true root, the difference between consecutive guesses gets smaller. As we approach a non-zero minimum or maximum, the difference will begin to get bigger again. This method isn’t fail safe either, as some functions may have a slope near vertical, causing the next guess to be very close to the current one.

Note that Newton’s method does not guarantee finding a root, even if one exists. Whether, and how fast you find a root depends on your initial guess.

```plaintext
program newton

! Disable implicit declarations (i-n rule)
implicit none
real(8) :: x, y, slope, tolerance

write (*,*) 'Enter initial guess for x:'
read (*,*) x
write (*,*) 'Enter the tolerance'
read (*,*) tolerance

! Compute first guess to prime the loop
y = x ** 2 - 1.0d0

do while ( abs(y) > tolerance )
! Recompute x, y and slope
slope = 2.0d0 * x
x = x - y / slope
y = x ** 2 - 1.0d0

! Debug output
write (*,*) y

enddo

write (*,*) x ! This is the solution within tolerance
end program
```

```c
#include <stdio.h>
#include <sysexits.h>
#define ABS(n) ((n) < 0 ? -(n) : (n))

int main(int argc, char *argv[])
{
    double x, y, slope, tolerance;
```
printf("Enter initial guess for x: ");
scanf("%lf", &x);

printf("Enter the tolerance: ");
scanf("%lf", &tolerance);

// Compute first guess to prime the loop
y = x * x - 1.0;

while ( ABS(y) > tolerance )
{
    // Recompute x, y and slope
    slope = 2.0 * x;
    x = x - y / slope;
    y = x * x - 1.0;

    // Debug output
    printf("%f\n", y);
}

printf("%f\n", x); // This is the solution within tolerance
return EX_OK;

23.13.3 Self-test

1. Modify the Newton’s method program so that it terminates when the difference between successive estimates is below the tolerance. Under what conditions would this produce a better estimate than the program that compares the y value to tolerance?

23.14 Code Quality

Indent

23.15 Performance

Order of operations

1. Eliminate entire loop if possible (better algorithm) sin(), cos(), sqrt(), strlcpy(), printf()
2. Reduce the number of iterations if possible (better algorithm) selection sort vs quick sort or heap sort
3. Remove unnecessary code inside the loop Often very obvious

   // Some bad code
#define MAX 1000000000

for (c = 0; c < MAX; ++c)
{
    /*
    * This if check is executed a billion times, but it doesn’t
    * need to be executed at all
    */
    if ( c == 0 )
    {
4. When there’s an if inside a loop, be suspicious

```c
// Some bad code
#define MAX 1000000000

for (c = 0; c < MAX; ++c)
{
    /*
     * This if check is executed a billion times, but it doesn’t
     * need to be executed at all
     */
    if ( c < SOME_VAL )
    {
        // case 1
    }
    else
    {
        // case 2
    }
}
```

```c
// Some better code
// case 1

for (c = 0; c < SOME_VAL; ++c)
{
    // case 1
}

for (c = SOME_VAL+1; c < MAX; ++c)
{
    // case 2
}
```

5. Combine small loops that iterate over the same range if possible so that the loop overhead is not duplicated

6. Break up large loops that might overwhelm the cache memory

### 23.15.1 Code Examples

### 23.15.2 Self-test

1. TBD
Chapter 24

Subprograms

24.1 Motivation

Most real-world programs are too large for the human mind to deal with all at once.

<table>
<thead>
<tr>
<th>Program</th>
<th>Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>ispell</td>
<td>10,000 lines</td>
</tr>
<tr>
<td>ape</td>
<td>13,000 + 13,000 twintk</td>
</tr>
<tr>
<td>pico</td>
<td>25,000</td>
</tr>
<tr>
<td>GNU Fortran Compiler</td>
<td>278,000</td>
</tr>
</tbody>
</table>

Table 24.1: Program Sizes

The only way to grasp programs of this scale is by designing and implementing them as a collection of smaller programs.

24.2 Modular Design

Most of the thought about how a task should be broken down is done in the design stage. Regardless of the complexity of a design, implementation should remain a fairly straightforward translation process.

24.2.1 Abstraction and Layers

Abstraction is the omission of detail for the sake of looking at the big picture. For example, an abstract is an overview of a book or paper that summarizes the whole story with very little detail.

Abstraction can have many levels. The highest level of abstraction covers the whole picture, and has the least detail.

When we go to lower levels of abstraction, we generally focus on only part of the picture. For example, we may have a more detailed abstract for each chapter of a book, and even more detailed abstracts for each section.

A table of contents is another form of abstraction, as is a review article, or an outline.

A top-down design is a multilevel view of something showing a highly abstract level, with very little concern for detail, as well as additional levels with increasing detail for smaller portions of the problem.

Stepwise refinement is the process of creating a top-down design, breaking down a large design into layers of increasing detail.

Consider a house as A simple example. At the most abstract level, as house can be viewed as a set of rooms. At the next level, each room consists of walls, a ceiling, and a floor. A floor typically consists of joists, subflooring, and a floor covering such as carpet or tile.
More detailed views of the house focus on smaller portions of the house, omitting more and more other components. This is necessary.

This is the approach we use to deal with anything that is too large and complex to comprehend, whether it be a house, a field of study, or an engineering problem. We choose a balance between breadth and detail, taking more of one and less of the other, depending on the needs of the moment.

The most intuitive way to visualize a top-down design is using a Tree structure view:

```
Earth
  Africa
  Asia
  Antarctica
  Australia
  N. America
    Canada
    Mexico
    USA
  S. America
  Europe
```

The problem with such a tree view is that it generally becomes too wide to represent on paper after 2 or three levels of refinement. The same structure can be rotated 90 degrees and represented as text using indentation to denote the level in the tree. This structure is more convenient to represent in documents, since it mostly grows vertically.

Top-down designs can be used to represent anything that can be broken down. A design can reveal the organization of physical objects to as much detail as we want, even down to subatomic particles.

Top-down designs can also show the structure of ideas and processes, which is how they are generally used in computer science. In this case, the tree or text diagram indicates a sequence of steps. On the tree, the sequence goes from left to right, and in the text it goes from the top down.

**Example 24.1 Ordering Pizza**

```
Order pizza
  Decide on toppings
    Talk to pizza eaters
      Talk to mom
      Talk to dad
      Talk to brother
        Wave at brother so he takes off headphones
      Talk to sister
        Wave at sister so she gets off the phone
    Write down toppings
  Decide on a pizza place
    Talk to pizza eaters
    Same details as above
  Decide on pick-up or delivery
    Talk to pizza eaters
    Same details as above
  Get phone number
  Check Internet
  Check phone book
  Call pizza place
```
One thing you may notice in the pizza example is that some of the modules we created when breaking down the process are used repeatedly. This is another benefit of breaking down a problem into layers and modules. In addition to simply making the problem manageable, if it turns out that some modules can be reused, the next step of implementing the solution is easier since you only have to implement a module once no matter how many times it is used.

Identifying and separating out modules that can be used in multiple places is called factoring the design (or the code). It is akin to reducing an expression such as $xy + xz$ to $x(y + z)$. It’s the same expression, but the latter only represents $x$ once. By factoring a program design, you ensure that each module of code will only be implemented once, which results in a smaller program.

Now let’s consider a problem for which we can easily implement a solution on a computer.

**Example 24.2 Sorting**

```
Sort list of numbers
  Get list
  Get number
  If not end of list, repeat
  Sort list
    Find smallest number in list
      Assume first is smallest
      Compare to next number
      If next is smaller, remember location
      Repeat compare until end of list
    Swap with first
      Copy first to temporary location
      Copy second to first
      Copy temporary location to second
    Repeat for remaining numbers
    Repeat until only one number remains
  Output sorted list
    Output number
    If not end of list, repeat
```

**Note** The key to stepwise refinement is avoiding the temptation to go into too much detail too quickly. At each step, we only break things down into a few, slightly smaller and less abstract units. We then tackle each of these smaller units in the same way, one at a time, until we have refined all the units at that level.

First level of refinement:

1. Sort list of numbers
   Get list
   Sort list
   Output sorted list

Second level of refinement:

1. Get list
   Get number
   Repeat until end of list
2. Sort list
   - Find smallest
   - Swap with first
   - Repeat previous two steps for remaining numbers
   - Repeat previous three steps until only one number remains

3. Output list
   - Output a number
   - Repeat until end of list

---

**Caution** An important part of using top-down design and implementation is to *stay focused on one layer and finish it before starting the next*. If you don’t, you will find yourself redoing an entire subtree when you alter the design at a higher level.

---

### 24.3 Subprogram Implementation

After a top-down design is developed, it’s now time to implement each of the modules in the top-down design.

The process we’re aiming for is to translate each module in the top-down design to a *subprogram*, an *independent* module within the larger program.

A *subprogram* is a module of code that is separated out from other code and made to operate independently.

Never think of a subprogram as part of another program. Every subprogram should be designed and implemented as a completely independent module that can be used in any program *without being modified*.

**Note** If the top-down design is well-developed, implementing the subprograms will be very easy. Each subprogram should be relatively small and easy to write without thinking about other subprograms.

If we run into trouble, we can go back and improve the top-down design. This is common, since the implementation phase will often reveal things that we didn’t foresee.

We want subprograms to be small, since the time it takes to *debug* a block of code is not proportional to the number of lines of code.

Debugging effort grows much faster than the numbers of lines as the size of the module increases. Hence, doubling the size of the module more than doubles the debugging effort. This is generally intuitive. Do you think it would be easier to write five 20-line programs, or one 100-line program?

The exact relationship between debugging effort and module size is next to impossible to determine exactly, and in fact varies depending on the project. However, generally speaking, it would be a concave-up increasing function. Suppose, for simplicity, that the relationship is parabolic:
debug time = $K_2 \times \text{size}^2$

Here, $K_2$ is another unknown constant that reflects the difficulty of debugging the code. The debug time for 1000 lines of code as one large module would be:

d debug time = $K_2 \times 1000^2 = 1,000,000 \times K_2$

Now suppose we could break the project into 10 independent modules of 100 lines each. The total amount of code is the same, but is the debug time the same?

d debug time = $10 \times K \times 100^2 = 10 \times K \times 10,000 = 100,000 \times K$

By breaking program into 10 independent modules, we have reduced the debug time by a factor of 10!

---

Note Note that we will only achieve a reduction in effort if the 10 modules are completely independent of each other. If you have to think about anything outside the module you’re working on, you cannot debug it independently.

---

This is what the use of subprograms does for us. No one on Earth is smart enough to comprehend a 100,000-line program as one large module, yet individual programmers often write programs much larger than this. The key is to break it into many smaller, independent modules during the design stage, and then coding and testing each module as a small, separate, independent program.

Writing a 100,000-line program will take a long time, even if it is broken into 2,000 subprograms averaging 50 lines each. Implementing 2,000 50-line subprograms is a lot of work, but it’s only a matter of time before you get it done. If anyone tried to tackle this as one single 100,000-line module, progress would stall quickly, and it would never get done.

### 24.4 Fortran Subprograms

Until now, the example programs presented contained only a main program. The main program actually is a subprogram. The only thing that distinguishes the main program from other subprograms is the fact that it is the first subprogram to run when a new process is created. For this reason, the main program is also called the entry point.
Within each subprogram there are *variable definitions* and *statements*, exactly like we have seen in our main programs.

Fortran offers two kinds of subprograms, which are implemented almost exactly the same way. The only difference between them is in how they are *invoked*, or *called*.

When a subprogram is called, the program "jumps" to the subprogram, runs it, and then returns to where the subprogram was called and continues executing the caller.

A *function* is a subprogram that is invoked by using it in an expression. Since a function call is part of an expression, it must return a value, which is used in the expression.

The main program calls other subprograms, which may in turn call other subprograms, and so on.

The *call tree* shows which subprograms call other subprograms. The call tree should have the same structure as the top-down design!

**Tree diagram for sort**

You have already seen examples of function calls such as `sin()`, `abs()`, `mod()`, etc. Consider the following statement:

\[
y = x \times 2 + \sin(x) - 5.6
\]

When this statement is executed, the program performs the following steps:

1. Compute \( x \times 2 \)
2. Call the `sin()` function
   (a) Send the *argument* \( x \) to the `sin()` function
   (b) Execute the `sin()` function
   (c) Return from the `sin()` function and return the sine of \( x \) to the statement containing the call to `sin()`.
3. Add the result of \( x \times 2 \) and the return value from `sin(x)`
4. Subtract 5.6 from the result of \( x \times 2 + \sin(x) \)
5. Store the result in memory address \( y \).

A Fortran *subroutine* is a subprogram that does not return a value, and is not called as part of an expression. A Fortran subroutine call is a separate statement by itself. You have seen examples of subroutine calls with `write`, `print`, and `read`. Consider the following program segment:

```
print *, 'Enter the radius of the circle:'
read *, radius
print *, 'The area of the circle is', PI * radius * radius
```

The sequence of events is as follows:

1. Call the `print` subroutine
   (a) Send the argument "Enter the radius of the circle:" to the `print` subroutine.
   (b) Jump to the `print` subroutine, which prints the string.
   (c) Return from the `print` subroutine and continue with the next statement after the `print`.
2. Call the `read` subroutine
   (a) Pass the argument variable `radius`.
   (b) Jump to the `read` subroutine, which reads input from the keyboard, converts it to the binary format appropriate for the variable `radius`, and stores the value in the memory location that `radius` represents.
   (c) Return from the `read` subroutine.
3. Compute \( \Pi \times radius \times radius \)
4. Call print

(a) Pass arguments 'The area of the circle is' and the value of the expression $\pi \times \text{radius} \times \text{radius}$ to the print subroutine.
(b) Jump to the print subroutine, which prints the arguments.
(c) Return from print subroutine.

24.5 **Internal vs External Subprograms**

Fortran functions and subroutines can be *internal* (part of another subprogram) or *external* (separate from all other subprograms). Internal subprograms are not independent from the subprogram that contains them, and therefore don’t do much to make the program modular and reduce debugging effort. They share variables with the subprogram that contains them and can only be called by the subprogram that contains them. They reduce typing time in exchange for increased debugging time and decreased reusability, which is usually a bad tradeoff.

External subprograms are completely independent from other subprograms. They can be written and debugged separately and even placed in a library for use by other programs.

All subprogram examples in this text will be external.

The C language does not support internal subprograms.

24.6 **Defining Your Own Fortran Functions**

Fortran has many intrinsic functions like sin(), sqrt(), etc. for common mathematical operations, but no language can provide a function for everything a programmer might want to compute.

Hence, Fortran allows us to define our own functions and even make them available for use in other programs.

A Fortran function definition looks almost exactly like the main program. There are only a few subtle differences. The general form of a function definition is as follows:

```fortran
! Define function
function function_name(argument variable list)
    data type :: function_name
    implicit none
    ! Define arguments
    data type, intent(in|out|inout) :: arg1, arg2, ...
    ! Define local variables
    data type :: var1, var2, ...
    ! Function body
    ! Return value by assigning to function name
    ! Every function MUST do this before returning to the caller!
    function_name = some-expression
end function function_name
```

Below is a complete program with a user-defined function. In addition to the function definition, the program contains a subprogram interface for the function, which is explained in Section 24.12.1.

```fortran
!-----------------------------------------------------------------------
! Program description:
! Program to print a table of integer powers
```
! Subprogram interfaces
module subprograms
    interface
        function power(base, exponent)
            real(8) :: power
            real(8) :: base
            integer :: exponent
        end function
    end interface
end module
! Main program body
program function_example
    use subprograms
    ! Disable implicit declarations (i-n rule)
    implicit none
    ! Variable definitions
    integer :: exponent
    ! Statements
    print *, 'Powers of 2'
    do exponent = 0, 10
        ! Will not work with 2.0 instead of 2.0d0
        print *, '2 ^ ', exponent, ' = ', power(2.0d0, exponent)
    enddo
    print *, ''
    print *, 'Powers of 3'
    do exponent = 0, 10
        ! Will not work with 3.0 instead of 3.0d0
        print *, '3 ^ ', exponent, ' = ', power(3.0d0, exponent)
    enddo
end program

! Description:
! Compute base ** exponent for any non-negative integer exponent
!
! Arguments:
! base: The real base of base ** exponent
! exponent: The non-negative integer exponent
!
! Returns:
! base ** exponent
!
! Modification history:
! Date Name Modification
! 2011-03-23 Jason Bacon Begin


```fortran
function power(base, exponent)
    ! Disable implicit declarations (i-n rule)
    implicit none

    ! Function type
    ! Factorials grow beyond the range of a 32-bit integer very quickly
    ! so we use a data type with greater range
    real(8) :: power

    ! Dummy variables
    real(8), intent(in) :: base
    integer, intent(in) :: exponent

    ! Local variables
    integer :: x

    ! Compute n!
    power = 1.0
    do x = 1, exponent
        power = power * base
    enddo
end function
```

### 24.7 Defining Your Own Subroutines

A subroutine is structured much like a function, except that it does not have a return value. Calling custom-written subroutines also differs slightly from calling intrinsic subroutines like `write`, `print`, and `read`, in that the arguments are enclosed in parentheses and we use the keyword `call` before the subroutine name.

```fortran
!-----------------------------------------------------------------------
! Program description:
! Swap two integer values
!-----------------------------------------------------------------------

!-----------------------------------------------------------------------
! Modification history:
! Date    Name    Modification
! 2011-03-23 Jason Bacon Begin
!-----------------------------------------------------------------------

module subprograms
    interface
        subroutine swap(num1, num2)
            integer, intent(inout) :: num1, num2
        end subroutine
    end interface
end module

! Main program body
program subroutine_example
    use subprograms

    ! Disable implicit declarations (i-n rule)
    implicit none

    ! Variable definitions
    integer :: x, y
```
! Statements
print *, 'Enter x and y on one line:'
read *, x, y
call swap(x, y)
print *, 'x = ', x, ', y = ', y
end program

!-----------------------------------------------------------------------
! Description:
! Swap the contents of two integer variables
!
! Arguments:
! num1, num2: References to the variables to swap
!-----------------------------------------------------------------------

!-----------------------------------------------------------------------
! Modification history:
! Date Name Modification
! 2011-03-23 Jason Bacon Begin
!-----------------------------------------------------------------------

subroutine swap(num1, num2)
    ! Disable implicit declarations (i-n rule)
    implicit none

    ! Dummy variables
    integer, intent(inout) :: num1, num2

    ! Local variables
    integer :: temp

    temp = num1
    num1 = num2
    num2 = temp
end subroutine

24.8 C Subprograms

In C, all subprograms are functions. Recall that C does distinguish between assignment statements and subroutine calls as Fortran
does. Any expression followed by a semicolon is a valid statement in C.

Following the minimalist philosophy, C also has no intrinsic (built-in) functions. Commonly used functions such as sin(), sqrt(),
etc. are provided by libraries, archives of functions written in C (or any other compiled language).

The basic syntax of a C function definition is as follows:

```c
return-type function-name(argument variable definitions)
{
    local variable definitions

    body

    return [value];
}
```

The return type can be any basic data type described in Table 20.1 or a pointer (discussed in Chapter 25).
It is also possible to define a function that does not return a value by giving it a return type of `void`. However, most C functions do return a value of some sort. Functions that do not compute a result such as a square root generally return a status value to indicate whether the function completed successfully.

Since any expression followed by a semicolon is a valid statement in C, it is acceptable (although not a good idea) to ignore the return value of some functions.

For example, the library function `printf()` returns the number of characters printed, or a negative value if an error occurred. We often ignore the return value:

```c
printf("The square of %f is %f\n", n, n * n);
```

However, errors can occur on output due to a disk being full or a network connection being lost, so it’s a good idea to check:

```c
if ( printf("The square of %f is %f\n", n, n * n); < 0 )
{
    // printf() failed. Handle the error as gracefully as possible.
}
```

The `scanf()` function returns the number of input items successfully read. It is important to check for success, since input errors are common.

```c
if ( scanf("%d %d", &rows, &cols) != 2 )
{
    // scanf() failed. Handle the error as gracefully as possible.
}
```

**Note** Only functions that absolutely cannot fail should be defined with a `void` return type. All others should return a status value so that the caller can detect errors and take appropriate action.

Below is a sample C program showing a definition of a `power()` function. In addition to the function definition, the program contains a *prototype* for the function above `main()`, which is explained in Section 24.12.1.

```c
/***************************************************************************
* Description:
* Program to print a table of integer powers
* *
* Arguments:
* None.
* *
* Returns:
* See sysexits.h
* *
* History:
* Date Name Modification
* 2013-08-12 Jason Bacon Begin
***************************************************************************/

#include <stdio.h>
#include <sysexits.h>

// Prototypes
double power(double base, int exponent);

int main(int argc,char *argv[])
{
    // Variable definitions
    int exponent;
```
// Statements
puts("Powers of 2");
for (exponent = 0; exponent <= 10; ++exponent)
    // Will not work with 2.0 instead of 2.0d0
    printf("2 ^ %d = %f\n", exponent, power(2.0, exponent));
puts("Powers of 3");
for (exponent = 0; exponent <= 10; ++exponent)
    // Will not work with 3.0 instead of 3.0d0
    printf("3 ^ %d = %f\n", exponent, power(3.0, exponent));
return EX_OK;
}

/********************************************************/
* Description:
* Compute base ** exponent for any non-negative integer exponent
* Arguments:
* base: The real base of base ** exponent
* exponent: The non-negative integer exponent
* Returns:
* base ** exponent
* History:
* Date Name Modification
* 2013-08-12 Jason Bacon Begin
*********************************************************/
double power(double base, int exponent)
{
    // Local variables
    double p;
    // Compute n!
    p = 1.0;
    while ( exponent-- >= 1 )
        p *= base;
    return p;
}

24.9 Scope

All variables defined within an external subprogram exist only in that subprogram. In other words, the scope of a variable in an external subprogram is confined to the subprogram in which it is defined.

The limited scope of variables provides another benefit to programmers: it allows the programmer to reuse variable names in different subprograms. For example, in the Fortran example program above with the power function, both the main program and the function have a variable called exponent. The x in the main program and the example in the function are different variables, i.e. they represent different memory locations, and can therefore contain separate values. Another way of saying this is that each subprogram has its own name space. The names of variables in one subprogram are kept separately from the names of variables in other subprograms.
24.10 Scope

Variables defined in C and Fortran are in scope (accessible) from where they are defined to the end of the block in which they are defined.

Most variables are defined at the beginning of a subprogram, so they are accessible to all statements in that subprogram and nowhere else.

In C, we can actually define variables at the beginning of any block of code. In the code segment below, the variable area is in scope only in the if block.

```c
#include <stdio.h>
#include <math.h>  // M_PI
#include <sysexits.h>  // EX_ *

int main()
{
    double radius;
    printf("Please enter the radius: ");
    scanf("%lf", &radius);
    if ( radius >= 0 )
    {
        double area;
        area = M_PI * radius * radius;
        printf("Area = %f\n", area);
    }
    else
    {
        fputs("Radius cannot be negative.\n", stderr);
        return EX_DATAERR;
    }
    return EX_OK;
}
```

Note If two variables by the same name are in scope at the same time, the one with the narrower scope is used.

24.10.1 Self-test

1. Define "scope".
2. What is the scope of a variable defined at the beginning of a function?

24.11 Storage Class

In addition to the data type, which indicates the binary format of the data, each variable also has a storage class, which indicates where in memory the data are stored and how that memory is managed.

24.11.1 Auto

The default storage class for local variables (variables defined inside the main program or inside any other subprogram) is auto. The term "auto" comes from C, but the same concept applies to Fortran and most other languages.
Auto variables use memory space on the system stack. When a program enters a new block of code where there are variable definitions, it allocates another block of memory on top of the stack, on top of already existing variables that have been allocated previously.

When the program exits that block, the space allocated for auto variables is immediately released. I.e., the top of the stack reverts to what it was before the block was entered.

The stack:

```
+-------------------------------------------------------+
| Auto variables in subprog2, called by subprog1 |
+-------------------------------------------------------+
| Auto variables in subprog1, called by main program |
+-------------------------------------------------------+
| Auto variables in main program |
+-------------------------------------------------------+
```

This strategy is very efficient, since allocation and deallocation are simple, and the minimum amount of stack space is in use at any given time.

Since auto variables are allocated when the code block is entered, initializers on auto variables cause the variable to be reinitialized every time the code block is entered, as if the variable definition were immediately followed by an assignment statement.

### 24.11.2 Static

Static variables are allocated at compile-time and remain allocated throughout the existence of a process.

The static storage class is the default for global variables in C, i.e. variables defined outside of any function including the main program. (Again, global variables should generally not be used, since they cause side-effects, which make the program difficult to debug.)

In C, local variables can be defined as static by adding the static modifier to the definition:

```c
static double radius;
```

Fortran, being a somewhat more abstract language than C, uses the `save` modifier to indicate static storage class. This indicates that the variable’s content should be saved across subprogram calls. It says more about how the variable is used than how it works behind the scenes.

```fortran
real(8), save :: radius
```

Since a static variable defined this way remains allocated throughout the life of the process, it retains its value after the subprogram exits and will have the same value next time the subprogram is called.

Recall that auto variables are deallocated when the program exits the block in which they are defined, so they may contain garbage left over from other uses of that memory location (e.g. auto variables in other code blocks) the next time the block is entered.

Since static variables are allocated at compile-time, initializers on static variables cause the variable to be initialized only when the process is first created.

### 24.11.3 Register

The C language also has a a register storage class, which attempts to keep the variable’s content in a CPU register at all times. Since CPU registers are very few, this cannot be guaranteed.

In theory, this can greatly improve performance, since a register can be accessed within a single clock cycle, while memory access may require waiting many clock cycles.

However, modern compilers have intelligent optimizers that can use precious CPU registers to temporarily cache many different variables, each when it matters most. This strategy is far more effective in improving program performance than keeping a few specific variables in registers at all times.

Hence, the register storage class is widely regarded as obsolete.
24.11.4 Dynamic

Storage space for variables can also be manually allocated and freed by the programmer. Memory allocated this way uses another area of memory known as the heap. We will discuss this in detail in Section 26.4.

24.11.5 Self-test

1. What are the 4 storage classes in C? Which one does Fortran not support?
2. When is memory allocated for an auto variable? When is it freed?
3. When is memory allocated for a static variable? When is it freed?
4. Why would we use a static variable in a subprogram?
5. When should we use the register storage class?

24.12 Argument Variables

As you can see, the structure of a subprogram is exactly like the structure of the main program. It really is a little program in and of itself.

The concept of arguments is new however, and requires some explanation.

The variables base and exponent in power() are called argument variables.

Fortran argument variables don’t have memory locations of their own, which is why they are also called dummy variables. Rather, they act as aliases for the arguments that are passed to the function by the calling subprogram.

C argument variables do have memory addresses of their own, so they are effectively like any other local variable, except that they are initialized to the value of the argument sent by the calling subprogram.

Note It makes no difference whether the name of an argument variable is the same as or different from the argument in the calling subprogram that it is representing. An argument variable is in the name space of the subprogram, and variables in the caller’s name space are unrelated. Furthermore, an argument from the caller need not be a variable: It could be a constant, or even an expression.

\[
y = \text{power}(3.0d0, x + 5)
\]

\[
y = \text{power}(3.0, x + 5);
\]

24.12.1 Type Matching

The first argument variable in a subprogram’s argument list represents the first argument from the caller, and so on. For example, the 3.0 above is represented by base and x + 5 is represented by exponent.

Unlike some intrinsic functions and subroutines, the argument types of our own functions and subroutines are not flexible. For example, the sin() function can take real, real(8), complex, or double complex arguments, and will return the same type. Our power function, on the other hand, must get a 32-bit floating point value for the first argument (base) and an integer for the second (exponent).

It is imperative that the data type of the argument in the caller is exactly the same as the data type of the argument variable. If the argument variable is an integer, then the subprogram will interpret the argument as a 32-bit two’s complement value, regardless of what type of argument was sent by the caller. For example, consider the following call to power():

\[
y = \text{power}(2.0d0, 3.0)
\]
The second argument, 3.0, is a Fortran real, and therefore formatted in binary as a 32-bit IEEE floating point value. Inside the power function, however, this value of 3.0 is represented by the integer argument exponent, and therefore the bits are interpreted as a 32-bit integer value.

The actual binary representation of 3.0 in 32-bit IEEE format is 01000000010000000000000000000000. Interpreted as a 32-bit integer value, this binary pattern is $2^{22} + 2^{30}$, or 1,077,936,128. Could this throw off the results of your program slightly?

To avoid problems with type matching, we should define an interface for each subprogram in Fortran or a prototype in C. A subprogram interface or prototype defines how to interact with the subprogram (how many arguments it takes, what data type each argument must be, and what data type of value it returns). It allows the compiler to validate all arguments passed to a subprogram and issue an error message when there is a mismatch. It also shows what type of value a function returns so that the compiler can use it appropriately.

The reason this is necessary is that C and Fortran use a one pass compiler, which reads through the source code from top to bottom only once. Hence, if a subprogram is called earlier in the program than it is defined, the compiler will not yet have seen the definition and will therefore not know what types of arguments it takes or what type of value it returns. By providing an interface or prototype before the first call to a subprogram, we give the compiler all the information it needs to verify the call to the subprogram and issue errors or warnings if necessary.

An interface or prototype is essentially an exoskeleton of a subprogram. It is identical to the subprogram with the local variables and statements removed. It defines how to communicate (interface) with the subprogram, but not how it works internally.

The most convenient way to create Fortran interfaces is to copy and paste the subprogram into a module, "gut" it, and use that module in any subprogram that calls the subprogram. This way, we need only write out the interface for each subprogram once.

The easiest way to create a C prototype is by copying and pasting the function header and adding a semicolon.

There is also a free program called cproto that generates prototypes directly from C source code.

### 24.12.2 Argument Passing Details

There are basically two ways to pass arguments to a subprogram in any language. Either we can send a copy of the value (pass by value), or, since every value is stored somewhere in memory, we can send its address (pass by reference).

**Pass by Value**

When passing by value, the receiving argument variable in the subprogram is not a dummy variable, but is instead like any other local variable. It has its own memory location, which receives a copy of the argument value passed from the caller. This is how all arguments are passed in C.

**Example 24.3 Sample C Code Passing by Value**

```c
x = 5;
y = power(2.0, exponent);
```

In Example 24.3, the argument variable base gets a copy of the argument 2.0, and the argument variable exponent gets a copy of the value of the argument exponent in the caller.

Since the variable exponent in power() represents a different memory address than exponent in the caller, changes to the variable exponent within the power function have no effect on exponent in the caller. Arguments passed to functions by value are therefore protected from side effects, inadvertent changes to their value. This is usually what we want when calling a subprogram. Imagine how annoyed you would be if you called sin(angle) and the sin() function changed the value of angle!
Pass by Reference

When an argument is passed by reference, the address of the data is sent to the subprogram rather than a copy of the value. That is, each time a subprogram is called, the argument variable in the subprogram is assigned the same address as the argument sent by the caller.

All simple arguments in Fortran are passed by reference.

```fortran
exponent = 5
y = power(2.0d0, exponent)
```

As Table 24.3 shows, when the power function is called, base assumes the address of the constant 2.0d0, and exponent assumes the address of the variable exponent in the caller.

If we called `power(a, b)`, then base would assume the address of `a`, and exponent would assume the address of `b`.

As a result of this, changes to the variable exponent in the power function would alter the value of exponent in the subprogram that calls `power(2.0d0, exponent)`, and will alter the value of `b` in the subprogram that calls `power(a, b)`.

This is not always desirable, so Fortran provides a way to protect arguments from alteration even though they are always passed by reference. This feature is discussed in Section 24.12.3.

In C, all arguments are passed by value. If we want a function to know the address of a variable, i.e. we want to pass it by reference, we must explicitly pass the address of the variable. You have already seen this when using the `scanf()` library function.

```c
scanf("%d", &x);
```

The `scanf()` function needs the address of the variable so that it can store the input value in it.

Another subprogram that needs the address of its arguments is the Fortran swap subroutine shown earlier. The implementation of an equivalent C function is shown in Chapter 25.

### 24.12.3 Intent

Since all simple variables in Fortran are passed by reference, the arguments passed to them could be vulnerable to side-effects.

If the argument passed by reference to a subprogram is a variable, its value could be altered, which means that the caller is not independent from the subprograms it calls, and hence the program will be harder to debug.

If the argument passed is not a variable (it is a constant or an expression), then it does not make sense to alter the argument variable, since it does not represent a variable in the caller. In this case, Fortran will produce an error.

Fortunately, Fortran provides a mechanism for protecting arguments, so that we can pass variables as arguments without worrying that they could be modified, and we can pass constants and expressions without causing an error.

Fortran handles this using a modifier in the variable definition called *intent*. The intent of an argument variable is one of the three values *in*, *out*, or *inout*. 

<table>
<thead>
<tr>
<th>Address</th>
<th>Name</th>
<th>Content</th>
</tr>
</thead>
<tbody>
<tr>
<td>1000</td>
<td>2.0</td>
<td>2.0</td>
</tr>
<tr>
<td>1008</td>
<td>exponent (caller’s namespace)</td>
<td>5</td>
</tr>
<tr>
<td>2000</td>
<td>base</td>
<td>2.0</td>
</tr>
<tr>
<td>2008</td>
<td>exponent (power’s namespace)</td>
<td>5</td>
</tr>
</tbody>
</table>

Table 24.2: Pass by Value

<table>
<thead>
<tr>
<th>Address</th>
<th>Name</th>
<th>Content</th>
</tr>
</thead>
<tbody>
<tr>
<td>1000</td>
<td>2.0d0, base</td>
<td>2.0</td>
</tr>
<tr>
<td>1008</td>
<td>exponent (caller), exponent (power)</td>
<td>5</td>
</tr>
</tbody>
</table>

Table 24.3: Pass by Reference
Caution If no intent is specified, the default is inout. This could be dangerous to your debugging efforts, as it leaves arguments open to side-effects! The intent of every argument variable should therefore be specified explicitly.

If the intent is in, then the argument variable is meant to take in information from the caller only. The value received from the caller is used by the subprogram, but cannot be modified, i.e. it is read-only. This is the type of argument used by a function like sin() or a subroutine like write.

If the intent is out, then the value received from the caller in the argument variable is not used, but the subroutine alters the value of the argument variable in order to send information back to the caller. The arguments to the intrinsic subroutine read are intent(out), since the read statement has no use for the values in the variables before read is called, but its purpose is to place new values in them. This is the type of argument used by a subroutine like read.

If the intent is inout, then the argument variable is meant to take in information from the caller and send information back. Hence, the subprogram will receive a value in the argument variable that it both uses and modifies for the caller. This situation is fairly rare. Usually, arguments are used to either receive information from the caller, or to send information back, but not both.

Note that all arguments to functions are intent(in), since the purpose of a function is to compute and return a single value through the return value. Subroutines generally use a mixture of intent(in) and intent(out) arguments.

24.13 Top-down Programming and Stubs

When designing and implementing something, it is generally preferable to start at the most abstract level, and work down from there. For example, when designing and building a house, we prefer to start by deciding what kind of house we want. This in turn helps us determine how many floors, the rooms on each floor, and eventually what kind of building materials we’ll need. This is the top-down approach.

The bottom-up approach begins with the materials we have available and determines the higher level features based on what’s possible with what we have. For example, if we start by deciding that our house will be made of snow, mud, or straw, this will strongly influence the type of house we end up with.

The top-down approach doesn’t prevent us from using pre-existing components at any level of the design or implementation. It merely allows the design to dictate the components, rather than vice-versa.

These principals apply to computer programs as well as to houses. In order to end up with a program that closely matches our needs, it is preferable to begin at the highest level of abstraction (what the program does and how is is used), and let that determine how the more detailed levels are designed and implemented.

In the implementation stage, the top-down approach involves writing the main program first, along with a stub for each subprogram called by main. A stub is simply an empty subprogram which has a complete interface, or signature, but no real code inside. The interface, or signature, refers to the argument list and return value. (These are the only features of a subprogram that other subprograms should be concerned with.)

The program consisting of a complete main calling one or more stubs can be compiled to weed out the syntax errors, and also tested to ensure that the subprogram interfaces work properly. A testable stub will return a value which indicates that all arguments were received. For example, consider the following program:

```fortran
module subprograms

!-----------------------------------------------------------------------
! Program description:
! Program to print a table of integer powers
!-----------------------------------------------------------------------

!-----------------------------------------------------------------------
! Modification history:
! Date  Name    Modification
! 2011-03-23 Jason Bacon Begin
!-----------------------------------------------------------------------

module subprograms
```

interface
  function power(base, exponent)
    real(8) :: power
    real(8) :: base
    integer :: exponent
  end function
end interface
end module

! Main program body
program function_example
  use subprograms

! Disable implicit declarations (i-n rule)
implicit none

! Variable definitions
integer :: x

! Statements
print *, 'Powers of 2'
do x = 0, 10
  ! Will not work with 2.0 instead of 2.0d0
  print *, '2 ^ ', x, ' = ', power(2.0d0, x)
enddo

print *, ''
print *, 'Powers of 3'
do x = 0, 10
  ! Will not work with 3.0 instead of 3.0d0
  print *, '3 ^ ', x, ' = ', power(3.0d0, x)
enddo
end program

!-----------------------------------------------------------------------
! Description:
! Compute base ** exponent for any non-negative integer exponent
!
! Arguments:
! base: The real base of base ** exponent
! exponent: The non-negative integer exponent
!
! Returns:
! base ** exponent
!-----------------------------------------------------------------------

function power(base, exponent)
! Disable implicit declarations (i-n rule)
implicit none

! Function type
! Factorials grow beyond the range of a 32-bit integer very quickly
! so we use a larger data type
real(8) :: power

! Dummy variables
This program contains a stub for power(), which compiles, executes, and returns a value indicating that the arguments base and exponent were received properly. The code to actually compute the power will be added later when implementing the second level of abstraction. The stub allows the implementation of the main program to be tested before beginning the implementation of the subprograms called by the main program. After testing with the stub, we now know that the main program works properly. If we encounter errors while implementing power() we don’t need to look for bugs in the main program, so the debugging process will be much simpler.

24.14 C Preprocessor Macros

24.15 Recursion

Foirtran 77 does not allow recursion by default.

24.16 Creating your own Libraries

Static:

```
ar -rs libmystuff.a *.o
```

Dynamic:

```
cc -shared -o libmystuff.so *.o
```

```
cc myprog.c -o myprog -Lparent-dir-of-library -lmystuff
```

24.17 Self-test

1. Why are subprograms essential to software development?

2. What is abstraction?

3. Without looking at the lecture notes, write a top-down design outlining the procedure for sorting a list of numbers. Use stepwise refinement to draw the design one layer at a time.

4. Write a main program that calls a stub for computing a power of any real base and any real exponent. Verify that the subprogram receives all arguments correctly and returns a result correctly.
Chapter 25

Memory Addresses and Pointers

Virtually every value used in a program is stored in memory and has a memory address.

We sometimes need to use the memory address of a variable instead of the data it contains. For example, the Fortran read subroutine and the C `scanf()` function both need to know the memory address of a variable in order to place input data in it.

Since Fortran passes arguments by reference, Fortran subprograms naturally know the address of all the arguments passed to them.

Since C passes arguments by value, we need to explicitly pass the address of a variable to any function that needs to know it.

```c
if ( scanf("%d %d", &rows, &cols) != 2 )
```

So how do functions like `scanf()` make use of the addresses passed to them?

They use *pointer variables*. A pointer variable is a variable that contains a memory address rather than data.

We can assign a pointer variable the address of any other variable of the appropriate data type. We can then access the data in the variable pointed to using what is called *indirection*.

```c
int rows = 2,
    cols = 3;
int *ptr; // Holds the address of an int variable
ptr = &rows;
printf("%d\n", *ptr); // Indirect access to the value in rows, prints 2
ptr = &cols;
printf("%d\n", *ptr); // Indirect access to the value in cols, prints 3
```

Functions that need to modify a variable in the caller must know the address of that variable. We’ve already seen that we can pass the address from the caller using the & operator:

```c
scanf("%d %d", &rows, &cols);
```

To receive such an address in the function, we simply define the argument as a pointer. Consider the simple task of swapping the values of two variables:

```c
temp = first;
first = second;
second = temp;
```

We might try to implement this as a function for convenience:

```c
void swap_ints(int first, int second)
{
```
```c
int temp;

temp = first;
first = second;
second = temp;
}
swap(a, b);
```

Knowing that C passes arguments by value, however, makes it apparent that this function is useless. The argument variables first and second are at different memory addresses than a and b, so swapping them has no effect on a and b.

In order to create a working swap function, we need to define the argument variables as pointers and pass the addresses of the arguments:

```c
void swap_ints(int *first, int *second)
{
    int temp;
    temp = *first;
    *first = *second;
    *second = temp;
}
swap(&a, &b);
```

Pointer variables have many additional uses, but their primary purpose in scientific programming is for passing addresses of data to functions.
Chapter 26

Arrays

26.1 Motivation

Write a program that reads in a list of 1000 numbers and prints them in the order they were entered. This is easy:

```c
int c;
double number;

for (c = 0; c < 1000; ++c)
{
    scanf("%lf", &number);
    printf("%f\n", number);
}
```

Now, write a program that reads a list of 1000 numbers from the standard input and prints them in reverse order.

In order to achieve this, the entire list must be stored in a way that it can be traversed backwards.

There are basically two places to store a list of values:

- In a file
- In RAM

Whether it’s better to store the list in a file or in RAM depends on how large the list is and how fast we need to be able to read the list backwards. Disks are much larger than RAM, so they will accommodate larger lists. RAM is many times faster than disk, however, and will therefore lead to a faster program.

26.2 Design vs. Implementation

As always, we need to separate design and implementation. Deciding whether to store our list in a file or in RAM is an implementation detail. The need to store it somewhere and read it backwards is a design detail.
Do not let thoughts about RAM and disk creep into your head while designing the solution to this problem. The design demands only that the solution remember every value entered. This is the important difference between the solutions for printing the list in forward or reverse order. To print them in forward order, the solution can read one value, print it, and forget it. To print them in reverse order, they must all be remembered until the last one is entered.

Conceptually, a one-dimensional list of data can be represented as a vector:

\[
[a_0, a_1, \ldots, a_n]
\]

Vectors can represent any one-dimensional collection of data, such as a simple list, coefficients of a point in space, or a polynomial (each \(a_i\) represents a coefficient).

\[2.5x^2 - 3.9x + 7.2 = [2.5, -3.9, 7.2]\]

The variables we’ve worked with until now, which hold a single value, are scalar, or dimensionless.

## 26.3 Array Basics

To implement the solution to printing 1000 numbers in reverse order, we could define 1000 variables, and use 1000 read and print statements:

```c
double num1, num2, ... num1000
scanf("%lf", &num1);
scanf("%lf", &num2);
... 
scanf("%lf", &num1000);
printf("%f\n", num1000);
... 
printf("%f\n", num2);
printf("%f\n", num1);
```

```c
double precision :: num1, num2, ... num1000
read *, num1
read *, num2
... 
read *, num1000
print *, num1000
... 
print *, num2
print *, num1
```

We now have a program of 2000 lines + the overhead code framing out the program, defining the variables, etc. This is clearly more effort than we want to put into a program, and the program is not at all flexible. Where would we be if we needed to do the same for a billion numbers?

An array is a variable that holds multiple values of the same type. Each value in the array is distinguished from the others using an integer index, also known as a subscript. The term subscript comes from mathematics, where we might specify one element from an array of values such as \(K_0, K_1, \ldots, K_n\).

In C, we make a variable into an array by providing the number of elements it contains in square brackets following the name. We then select an element by specifying the subscript in square brackets after the name. C subscripts always begin at 0 and end at the array size minus one.

```c
#define LIST_SIZE 1000
double list[LIST_SIZE];
```
In Fortran, we make a variable into an array by providing the number of elements it contains in parentheses following the name. We then select an element by specifying the subscript in parentheses after the name. By default, Fortran subscripts begin at 1, but we can control this.

```fortran
module constants
  integer, parameter :: LIST_SIZE=1000
end module

double precision :: list(LIST_SIZE)
read *, list(1)
read *, list(2)
... 
read *, list(LIST_SIZE)
print *, list(1)
print *, list(2)
print *, list(LIST_SIZE)
```

An array makes it much easier to allocate the space for 1000 double precision values, but we still have 2000 statements to read the list and print it backwards. Can we do better?

The only restriction on subscripts is that the must be integers. They may be constants, variables, or complicated expressions. As long as the value of the subscript is within the range of subscripts for the array, there is no problem.

Most commonly, subscripts are a simple variable which is controlled by a loop that traverses all valid subscripts for the array:

Variables used to subscript an array must have enough range. A Fortran integer(1) or C char variable have a maximum value of +127, so they cannot be used as subscripts for an array of 1000 elements. A Fortran integer (integer(4)) variable has a maximum value of $2^{31}-1$ (a little over 2 billion), which is enough for most arrays. It is possible on modern computers to have arrays with more than 2 billion elements, so we may sometimes need to use integer(8).

The C header files define an unsigned integer type called `size_t` which has the same number of bits as a memory address on the underlying hardware. Hence, it is guaranteed to be able to handle an array of any size. This data type should be used for virtually all array subscripts in C.

```c
#define LIST_SIZE 1000

double list[LIST_SIZE];
size_t c;
for (c = 0; c < LIST_SIZE; ++c)
  scanf("%lf", &list[c]);
for (c = LIST_SIZE-1; c >= 0; --c)
  printf("%f", list[c]);
```

```fortran
module constants
  integer, parameter :: LIST_SIZE=1000
end module
```
double precision :: list(LIST_SIZE)
integer :: index

do index = 1, LIST_SIZE
   read *, list(index)
endo

do index = LIST_SIZE, 1, -1
   print *, list(index)
endo

In Fortran, if we want to define multiple arrays of the same size, we can use the dimension modifier instead of specifying the dimension for every array variable.

double precision, dimension(LIST_SIZE) :: list1, list2, list3

Fortran also allows the programmer to choose any starting and ending subscripts desired. The default starting subscript is 1, so the following is equivalent to the examples above:

double precision :: list(1:LIST_SIZE)

For some arrays, it may not make sense to use a starting subscript of one. For example, if an array contains the probability of a driver in the U.S. having an accident based on their age, then subscripts less than 15 would be useless, since people under are not allowed to drive.

We should also specify the array dimensions using named constants to make the program more readable and easier to modify.

module constants
   integer, parameter :: MIN_AGE=15, MAX_AGE=120
end module

program insurance
   double precision :: accident_probability(MIN_AGE : MAX_AGE)
   integer :: age
   ...
   do age = MIN_AGE, MAX_AGE
      print *, accident_probability(age)
   endo
   ...
end program

#include <stdio.h>
#include <sysexits.h>
#define MIN_AGE 15
#define MAX_AGE 120

int main()
{
   double accident_probability[MAX_AGE - MIN_AGE + 1];
   size_t age;
   for (age = MIN_AGE; age <= MAX_AGE; ++age)
      printf("%f\n", accident_probability[age - MIN_AGE]);
   return EX_OK;
}
26.4 Dynamic Memory Allocation

In the old days of programming, arrays were always defined with a fixed, or static size. However, the amount of input to a program is rarely fixed. This means that static arrays must be defined to accommodate the largest possible inputs. For example, a program made to process lists of up to 1,000,000 elements must use an array of 1,000,000 elements, even when processing a list of 3 elements. This is a colossal waste of memory resources.

All modern languages allow the size of arrays to be determined at run-time, so they only use as much memory as needed. For this reason, static arrays should not be used anymore, unless it is an absolute certainty that the size of the array will not vary much.

More often than not, we don’t know how big our list is until run-time.

In C, an array name is actually a pointer. The array name always represents the address of the first element in the array. The only difference between an array name and a pointer variable in C is that we cannot change what the array name points to. I.e., an array name is a pointer constant rather than a pointer variable.

Pointer variables and array names are completely interchangeable, with one exception: An array name by itself cannot be on the left side of an assignment statement. Most importantly, we can use subscripts with pointer variables just like we do with array names.

Hence, to create a dynamically allocated array in C, we begin by defining a pointer variable instead of an array variable.

We then use the malloc() library function to allocate memory, which is prototyped in stdlib.h. The malloc() function returns the address of the allocated memory, or the constant NULL if the allocation failed.

The return type of malloc() is void *, so we should cast it to the pointer type of the array to avoid a compiler warning.

Lastly, note that malloc() takes the number of bytes to allocate as an argument, not the number of array elements. With malloc(), we generally use the C sizeof() operator, which returns the size of a variable or data type.

```c
#include <stdio.h>
#include <stdlib.h>
#include <sysexits.h>

int main(int argc, char *argv[]) {
    double *temperatures;
    size_t num_temps,
    c;
    printf("How many temperatures are there? ");
    scanf("%zu", &num_temps); // %zu for size_t
    printf("%zu\n", num_temps);
    temperatures = (double *)malloc(num_temps * sizeof(*temperatures));
    if ( temperatures == NULL )
    {
        fprintf(stderr, "Cannot allocate memory for temperatures.\n")
        exit(EX_OSERR);
    }
    puts("Enter the temperatures separated by whitespace:");
    for (c = 0; c < num_temps; ++c)
        scanf("%lf", &temperatures[c]);
    // Careful here: c is unsigned, so it is always >= 0
    while (c-- > 0)
        printf("%f\n", temperatures[c]);
    // Always free memory as soon as possible after it’s used
    free(temperatures);
    return EX_OK;
}
In Fortran, we can indicate that the size of an array is to be determined later by adding the `allocatable` modifier and placing only a `:` in the dimension. We then use the `allocate` intrinsic subroutine to allocate the array at run-time.

```fortran
program allocate
    use iso_fortran_env
    implicit none

    double precision, allocatable :: temperatures(:)
    integer :: num_temps, allocate_status, i

    print *, 'How many temperatures are there?'
    read *, num_temps
    allocate(temperatures(1:num_temps), stat=allocate_status)
    if ( allocate_status /= 0 ) then
        write(ERROR_UNIT, *) 'Cannot allocate memory for temperatures.', &
        'Error code = ', allocate_status
        stop
    endif

    print *, 'Enter the temperatures one per line:'
    do i = 1, num_temps
        read *, temperatures(i)
    enddo

    do i = num_temps, 1, -1
        print *, temperatures(i)
    enddo
end program
```

We must provide an integer variable along with "stat=" to receive the status of the allocation attempt. If the memory for the array is allocated successfully, the status variable will be set to 0. If the allocation fails (there is not enough memory available to satisfy the request), it will be set to a non-zero value. Programs can be very sophisticated with the status codes returned. They may decide to request a smaller block, or deallocate something else and try again.

The very least we should do is stop the program. Failure to check the status of a memory allocation can cause incorrect output, which could be catastrophic in some cases.

Once the allocatable array is allocated, it can be used like any other array.

When the array is no longer needed, it should be `immediately` deallocated, to free up the memory for other uses.

```fortran
deallocate(temperatures)
```

### 26.5 Array Constants

Fortran includes the concept of an `array constant`. This can reduce the size of a program if an array must be initialized to a set of constant values.

```fortran
list = (/ 2, 4, 10 /)
```

Same as

```fortran
list(1) = 2
list(2) = 4
list(3) = 10
```

C does not have the same flexible array constant construct, but it does allow array initializers. In addition, when an initializer is used on an array, we can omit the array size, since it can be inferred from the initializer.

```fortran
double list[] = { 2, 4, 10 };
```
26.6 Static Array Initialization

26.6.1 The Fortran DATA Statement

In C, if we initialize a static array, the initialization occurs before the program begins running.

```c
static double list[] = { 2, 4, 10 ];
```

Without the static modifier, the array would be initialized at run time, when the block containing the variable definition is entered. With the static modifier, the initialization takes place at compile time, so the numbers are already in the array as soon as the process begins executing.

Another way to look at it is that variables are always initialized when they are instantiated, i.e. when their memory is allocated. Static variables are instantiated at compiled time and auto variables at run time, when they are needed.

Another way to initialize a Fortran array is using a data statement.

```fortran
integer :: list(3)
data list / 2, 4, 10 /
```

This differs from assigning the array an array constant in that the data statement is processed at compile-time, whereas the assignment is done at run-time. When the program begins execution, the values from a data statement will already be in the array. Since the data statement is not performed at run-time, it actually makes no difference where it is placed within the subprogram. It is processed and removed by the compiler, not translated to machine code.

You may need to assign values to an array at run-time, or even assign the array several different sets of values at different times. In this case, an assignment statement is necessary.

However, if the contents of an array will not change during program execution, then using a data statement will save execution time, since the data was loaded into the array at the same time the program itself was loaded into memory.

26.6.2 Look-up Tables

A look-up table is a list of precomputed values. If a value is expensive to compute (i.e. requires a loop) and frequently needed, it may be quicker to store precomputed values in an array and look them up instead. This only involves indexing the array (computing the address of an element) which is trivial (a few machine instructions).

Ideal candidates for look-up tables are mathematical functions with a small domain that require a loop to compute. Since we are storing precomputed results for all cases, a look-up table will generally require more memory than the code to compute results on-the-fly.

The factorial is one such function. Only a small number of factorials are within the range of typical integer or floating point types, and they must be computed iteratively.

To achieve a performance gain using a look-up table, we must use a Fortran data statement or a C static array. Assigning an array constant at run-time or initializing an automatic array is costly, since it is itself an implied loop.

```fortran
unsigned long fastfact(unsigned int n) {
    static unsigned long table[] = { 1ul, 1ul, 2ul, 6ul, 24ul, 120ul, 720ul, 5040ul, 40320ul, 362880ul, 3628800ul, 39916800ul, 479001600ul, 6227020800ul, 87178291200ul, 1307674368000ul, 20922789888000ul, 355687428096000ul, 6402373705728000ul, 121645100408832000ul };
    return table[n];
}
```
function fastfact(n)

    ! Disable implicit declarations (i-n rule)
    implicit none

    ! Function type
    integer(8) :: fastfact

    ! Argument variables
    integer, intent(in) :: n

    ! Local variables
    integer(8) :: table(0:19)

    fastfact = table(n)

    data table / 1_8, 1_8, 2_8, 6_8, 24_8, 120_8, 5040_8, 40320_8, &
    362880_8, 3628800_8, 39916800_8, 479001600_8, 6227020800_8, &
    87178291200_8, 1307674368000_8, 209227898880000_8, &
    355687428096000_8, 6402373705728000_8, 121645100408832000_8 /
end function

To generate the factorial data for the lookup table requires a program. Integers are limited to about 19!. Double precision floating point can represent up to 170!, but there’s another problem: Even 64-bit floating point systems are limited to about 16 significant figures. As soon as the factorial value reaches 17 significant figures, it gets rounded. All later factorials are computed from this rounded value, and so the round-off error grows rapidly from this point on. 170! has 305 significant digits, but at most 16 of them will be accurate if the value is computed using double precision. Output from a C program using double precision follows. Note that 171! is reported as infinity, since 64-bit IEEE format cannot represent it.

```
169! = 4.26906800900470559560e+304
170! = 7.25741561530800402460e+306
171! = inf
```

What we need to compute large factorials precisely is an arbitrary-precision integer package. Such a package works by concatenating integer values end-to-end, and processing these large integers in chunks (e.g. 64 bits at a time on a 64 bit computer). There are libraries available for C, C++, Java, etc. However, the Unix `bc` is quite convenient for this purpose, and includes a C-like scripting language. Below is a `bc` script that prints factorials. You can run it by saving it to a file (e.g. `fact.bc`) and running `bc fact.bc` at the Unix prompt.

```
f = 1
for (c = 1; c <= 170; ++c) {
    f *= c
}
quit
```

```
169! = 426906800900470559560e+304
170! = 725741561530800402460e+306
171! = inf
```

```
f = 1
for (c = 1; c <= 170; ++c) {
    f *= c
}
quit
```

```
169! = 426906800900470527493925188889956538068818636056736103849163411179\ 7755494218009285432397014271652653818230139905012221568248567907501\ 7796052357489459546484708413412076211998036035274015123788150487897\ 5040568419670360154453585622827477179746400268937258948624384000000\ 000000000000000000000000000000000
170! =
```
A modified version of the bc script can generate output that can be copied and pasted directly into a program. Note that this version requires GNU bc, which is not standard on all systems.

```c
fact = 1
for (c = 1; c <= 170; ++c) {
    print fact, ",
    fact = fact * c;
}
quit
```

### 26.7 Arrays as Arguments

Arrays can be passed as arguments to functions and subroutines. When doing so, we need to also send the dimensions of the array. We will need the size in order to properly set up loops inside the subprogram that access the array.

In C, a one-dimensional array argument need not be given a size. This allows a function to work with arrays of any size, so long as it is told the size of each array when it is called.

```c
#define MAX_TEMPS 1000
int main()
{
    double temps[MAX_TEMPS];
    print_list(temp, MAX_TEMPS);
    return EX_OK;
}

void print_list(double list[], size_t list_size)
{
}
```

Note that we pass an array to a function, we are not copying the array to the function’s argument variable. Passing an array itself by value would mean duplicating the entire contents of the array, which would be very inefficient for large arrays.

Remember that an array name in C is a pointer constant. It represents the address of the array, whereas a scalar variable name represents the value contained in it. When we use an array name as an argument to a function, we are passing the address of the array, much like we did explicitly with our swap() function example:

```c
int a, b;
swap(&a, &b);
```

In fact, we could declare the function argument variable as a pointer rather than an array. The following would be essentially identical to the example above:

```c
void print_list(double * const list, size_t list_size)
{
}
```
The trick to understanding the use of `const` with pointers is to read it backwards. The above says "list is a constant pointer to a double".

Since we are passing the address of an array, the array contents are not protected from side effects. When we pass an array to a function, the function may be able to modify it.

We can protect against this by declaring the argument variable as an array of constants:

```c
void print_list(const double list[], size_t list_size)
{
}
```

We can protect against this by declaring the argument variable as an array of constants:

```
void print_list(const double list[], size_t list_size)
{
}
```

Again, reading this backwards, we see that "list is a pointer to a double constant", meaning that the data it points to is constant. Since list is declared as an array, the address is also constant. We can neither make change where list points, nor change the content of what it points to.

If we inadvertently try to modify the contents of list[] in the print_list() function, we will now get a compiler error.

Fortran does require a size for an array argument. However, since arguments are never without a value, the array size argument can be used to declare the array in Fortran. Array dimensions can be variables, as long as they have a known value. We must also declare the size argument before the array, so that the compiler has seen it before it encounters the array declaration.

```fortran
program array_arguments
    double precision, allocatable :: temps(:)
    integer :: num_temps
    ...
    call print_list(temps, num_temps)
    ...
end program

subroutine print_list(list, list_size)
  ! Define list_size before list
  integer, intent(in) :: list_size
  double precision, intent(in) :: list(1:list_size)
  integer :: i
  do i = 1, list_size
      print *, list(i)
  enddo
end subroutine
```

### 26.8 Allocate and Subprograms

Unlike some other languages, an array allocated in a Fortran subprogram is automatically deallocated when the subprogram returns.
To some extent, this has the advantage of preventing memory leaks, where a programmer allocates more and more memory over time and forgets to deallocate all of it, resulting in a continual reduction in available memory. Automatically deallocating arrays does not completely eliminate memory leaks however, and some might argue that it leads to less disciplined programmers by allowing them to be less vigilant.

Automatically deallocating arrays when a subprogram returns is sometimes a disadvantage as well. There are situations where we want a subprogram to allocate memory which can later be used by the calling subprogram, or others. In Fortran, however, allocated memory is only accessible to the subprogram that allocated it, and other subprograms called by it after the allocation and before it returns. Consider the following read_list() implementation:

```fortran
! Main program body
program cant_do_this
  ! Disable implicit declarations (i-n rule)
  implicit none

  ! Variable definitions
  integer :: num_temps
  real(8), allocatable :: temps(:)

  ! Allocate an array for a list of temps and fill it from input
  ! Get back the list and the list size from read_list()
  call read_list(temps, num_temps)

  ! Print the list returned by read_list
end program

subroutine read_list(list, list_size)
  ! Disable implicit declarations (i-n rule)
  implicit none

  ! Dummy variables
  ! Define list_size first, since it is used to define list
  integer, intent(out) :: list_size
  real(8), intent(out), allocatable :: list(:)

  ! Local variables
  integer :: i, alloc_status

  read *, list_size
  allocate(list(1:list_size), stat=alloc_status)
  if ( alloc_status /= 0 ) then
    print *, 'Allocate failed. status = ', alloc_status
    stop
  endif

  do i = 1, list_size
    read *, list(i)
  enddo
end subroutine
```

The code above will compile, but will not work, because the array allocated inside the read_list() subroutine is deallocated when the subroutine returns. Hence, the array no longer exists after coming back to the main program from read_list().

The code below shows how this task must be accomplished in Fortran. The array must be allocated before calling read_list() and passed to read_list() to receive the input. The array can exist in the main program, in read_list(), and in other subprograms called by the main program.

```fortran
! Main program body
program this_one_works
  ! Disable implicit declarations (i-n rule)

  ! Variable definitions
  integer :: num_temps
  real(8), allocatable :: temps(:)

  ! Allocate an array for a list of temps and fill it from input
  ! Get back the list and the list size from read_list()
  call read_list(temps, num_temps)

  ! Print the list returned by read_list
end program
```
implicit none

! Variable definitions
integer :: num_temps, alloc_status
real(8), allocatable :: temps(:)

read *, num_temps
allocate(temps(1:num_temps), stat=alloc_status)
if ( alloc_status /= 0 ) then
   print *, 'Allocate failed. status = ', alloc_status
   stop
endif

! Allocate an array for a list of temps and fill it from input
! Get back the list and the list size from read_list()
call read_list(temps, num_temps)

! Print the list returned by read_list

end program

subroutine read_list(list, list_size)
   ! Disable implicit declarations (i-n rule)
   implicit none

   ! Dummy variables
   ! Define list_size first, since it is used to define list
   integer, intent(in) :: list_size
   real(8), intent(out) :: list(1:list_size)

   ! Local variables
   integer :: i

   do i = 1, list_size
      read *, list(i)
   enddo

end subroutine

In C, memory allocated by malloc() remains allocated until it is released by free(). This is part of the C philosophy of "trust the programmer". This philosophy often leads to programmers shooting themselves in the foot, but also allows programmers to easily do what they need to.

26.9 Sorting

One of the most common uses for arrays is sorting lists. Like reversing a list, sorting is inefficient to perform on disk, and easy and fast if done in memory. Of course, if the list is very large, we may not have enough RAM to hold it. In these cases, however, we can sort parts of the list into memory, and merge them later.

Sorting is a very well-studied problem, and many sorting algorithms have been developed. The most efficient algorithms are somewhat difficult to understand, so we will focus on the simpler ones in order to demonstrate the use of arrays more easily.

One of the simplest and most intuitive sorting algorithms is the selection sort. The design of the selection sort is as follows:

1. Read the list
2. Sort the list
   (a) Find the smallest (or largest) element in the list.
(b) Swap it with the first element.
(c) Repeat the previous two steps starting at the next element.
(d) Repeat the previous three steps until the entire list is sorted.

3. Print the sorted list

To help illustrate a clean implementation process, here is a framed-out first step with stubs for reading and printing the list:

```c
#include <stdio.h>
#include <stdlib.h>
#include <sysexits.h>

double *read_list(size_t *size);
void print_list(const double list[], size_t list_size);
void sort_list(double list[], size_t list_size);

int main(int argc, char *argv[])
{
    double *list;
    size_t list_size;
    list = read_list(&list_size);
    print_list(list, list_size);
    free(list);
    return EX_OK;
}

double *read_list(size_t *size_ptr)
{
    double *list;
    *size_ptr = 10;
    list = (double *)malloc(*size_ptr * sizeof(*list));
    return list;
}

void print_list(const double list[], size_t list_size)
```

After testing the stubs, we fill them out and test the completed read and print functions:

```c
#include <stdio.h>
#include <stdlib.h>
#include <sysexits.h>

double *read_list(size_t *size);
void print_list(const double list[], size_t list_size);
void sort_list(double list[], size_t list_size);

int main(int argc, char *argv[])
{
    double *list;
    size_t list_size;

    list = read_list(&list_size);
    print_list(list, list_size);
    free(list);
    return EX_OK;
}

/*
 * Input list size, allocate array, and read in list.
 */

double *read_list(size_t *size_ptr)
{
    size_t c;
    double *list;

    scanf("%zu", size_ptr);  // No & here, since size_ptr is a pointer
    list = (double *)malloc(*size_ptr * sizeof(*list));
    for (c = 0; c < *size_ptr; ++c)
       scanf("%lf", &list[c]);
    return list;

```
void print_list(const double list[], size_t list_size)
{
    size_t c;

    for (c = 0; c < list_size; ++c)
        printf("%f\n", list[c]);
}

Finally, we implement the sort function. Note that while we work on the sort function, we need not worry about anything else, since we know that the rest of the program is already tested.

#include <stdio.h>
#include <stdlib.h>
#include <sysexits.h>

double *read_list(size_t *size);
void print_list(const double list[], size_t list_size);
void sort_list(double list[], size_t list_size);

int main(int argc, char *argv[])
{
    double *list;
    size_t list_size;

    list = read_list(&list_size);
    sort_list(list, list_size);
    print_list(list, list_size);
    free(list);
    return EX_OK;
}

/*
 * Input list size, allocate array, and read in list.
 */

double *read_list(size_t *size_ptr)
```c
{
    size_t c;
    double *list;

    scanf("%zu", size_ptr); // No & here, since size_ptr is a pointer
    list = (double *)malloc(*size_ptr * sizeof(*list));
    for (c = 0; c < *size_ptr; ++c)
        scanf("%lf", &list[c]);
    return list;
}

void print_list(const double list[], size_t list_size)
{
    size_t c;

    for (c = 0; c < list_size; ++c)
        printf("%f\n", list[c]);
}

/*
 * Sort list using selection sort algorithm.
 */

void sort_list(double list[], size_t list_size)
{
    size_t start, low, c;
    double temp;

    for (start = 0; start < list_size - 1; ++start)
    {
        /* Find lowest element */
        low = start;
        for (c = start + 1; c < list_size; ++c)
            if (list[c] < list[low])
                low = c;

        /* Swap first and lowest */
        temp = list[start];
        list[start] = list[low];
        list[low] = temp;
    }
}

shell-prompt: ./selsort
3
6.0
2.1
9.4
2.100000
6.000000
9.400000
And the Fortran implemtation:
!-----------------------------------------------------------------------
! Description:
! Usage: sort input-file output-file
! Sort a list of real numbers in input-file, placing the results
! in output-file. Both input and output files contain one
! number per line.
! Arguments:
! input-file: file to be sorted
! output-file: file to receive the sorted list
! Modification history:
! Date Name Modification
! Apr 2010 J Bacon Start.
!-----------------------------------------------------------------------

module constants
! Global Constants
real(8), parameter :: &
 PI = 3.1415926535897932d0, &
 E = 2.7182818284590452d0, &
 TOLERANCE = 0.00000000001d0
end module constants

program selection_sort
! Import stuff from constants module
use constants
use ISO_FORTRAN_ENV
! Disable implicit declarations (i-n rule)
imPLICIT none
! Local variables
integer :: list_size, allocate_status
real(8), allocatable :: list(:)
! Get size of list
read *, list_size
! Allocate array for list
allocate(list(1:list_size), stat=allocate_status)
if (allocate_status /= 0) then
    print *, 'Error: Could not allocate array of size ', list_size
    stop
endif
! Read list
call read_list(list, list_size)
! Sort list
call sort_list(list, list_size)
! Output list
call print_list(list, list_size)
end program

!--- Description:
! Read a list of real numbers from the standard input to
! the array list. The input file contains one number per line.
subroutine read_list(list, list_size)
  ! Import stuff from constants module
  use constants

  ! Disable implicit declarations (i-n rule)
  implicit none

  ! Dummy variables
  integer, intent(in) :: list_size
  real(8), intent(out) :: list(1:list_size)

  ! Local variables
  integer :: i

  do i = 1, list_size
    read *, list(i)
  enddo
end subroutine

subroutine print_list(list, list_size)
  ! Disable implicit declarations (i-n rule)
  implicit none

  ! Dummy variables
  integer, intent(in) :: list_size
  real(8), intent(in) :: list(1:list_size)

  ! Local variables
  integer :: i

  do i = 1, list_size
    print *, list(i)
  enddo
end subroutine
 subroutine sort_list(list, list_size)
 ! Import stuff from constants module
 use constants

 ! Disable implicit declarations (i-n rule)
 implicit none

 ! Dummy variables
 integer, intent(in) :: list_size
 real(8), intent(inout) :: list(1:list_size)

 ! Local variables
 logical :: sorted
 integer :: c, low, start
 real(8) :: temp

 do start = 1, list_size
 ! Find low
 low = start
 do c = start+1, list_size
 if ( list(c) < list(low) ) then
 low = c
 endif
 enddo

 ! Swap with first
 temp = list(low)
 list(low) = list(start)
 list(start) = temp
 enddo
end subroutine

26.10 When to Use Arrays

Many people have a tendency to use the most complicated tools available to implement a solution. This is a bad tendency that stems largely from a desire to look or feel smart. Ironically, when we do this we really make ourselves look foolish, at least in the eyes of those who are wise enough to know better.

More sophisticated solutions cost more. They are harder to implement, consume more resources, are more prone to design and implementation errors, and more prone to failure.

As a simple example, consider the problem of reading a list and printing it back in forward order. This can be done with an array, but should it? Drawbacks of using an array:

- The size of the list is limited by available RAM.
• The program is more complicated.
• The program must traverse the list twice: Once to read it and again to print it.
• The program consumes vastly more memory than a program without an array.

In summary, the array version is bigger, slower, harder to write, and places more load on the computer running it. A programmer who uses an array to solve this problem is demonstrating poor judgment and a lack of understanding of the specifications. Good programmers always look for the simplest, most elegant solutions that minimize resource requirements.

The same argument can be made for a case where you need to add two vectors or matrices contained in files. We could inhale them into arrays, add corresponding elements, storing the results in a sum array, and finally dump the sum array to an output file. But this would be a foolish approach. We only need 3 scalar variables for this task. We can read one element at a time from each source file, add them, and immediately write the result to the sum file.

Whenever designing a solution to a problem, ask yourself if it is really necessary to retain large amounts of information. Retain only as much as necessary to make the solution correct and efficient.

### 26.11 Array Pitfalls

#### 26.11.1 Address Calculation

```fortran
double precision :: list(LIST_SIZE)
integer :: index
double precision :: tolerance
```

An array is a contiguous block of memory. Assuming the base address of `list` is 1000, the array would map into memory as follows:

<table>
<thead>
<tr>
<th>Address</th>
<th>Content</th>
</tr>
</thead>
<tbody>
<tr>
<td>1000-1007</td>
<td>list(1)</td>
</tr>
<tr>
<td>1008-1015</td>
<td>list(2)</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>8992-8999</td>
<td>list(1000)</td>
</tr>
</tbody>
</table>

Table 26.1: Memory Map of an Array

The address of an array element with index `i` is the base-address + (i - lowest-subscript) * (number of memory cells used by 1 element).

E.g., the address of `list(i)` = 1000 + (i-1) * 8.

<table>
<thead>
<tr>
<th>Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>address of list(1) = 1000 + (1-1) * 8 = 1000</td>
</tr>
<tr>
<td>address of list(2) = 1000 + (2-1) * 8 = 1008</td>
</tr>
<tr>
<td>...</td>
</tr>
<tr>
<td>address of list(1000) = 1000 + (1000-1) * 8 = 8992</td>
</tr>
</tbody>
</table>

#### 26.11.2 Out-of-bounds Errors

Most compilers don’t generate machine code that checks array subscripts. Validating the subscript on every array reference would slow down array access significantly.

The programmer must ensure that their code uses only valid array subscripts!
int list[5], c;
for (c = 0; c <= 5; ++c)
{
    list[c] = 0
    printf("%d\n", c);
}

integer :: list(5), i
do i = 1, 6
    list(i) = 0
    print *, i
enddo

This code will overrun the end of the array 'list'.
Assume the base address of list is 4000.

<table>
<thead>
<tr>
<th>Address</th>
<th>Content</th>
</tr>
</thead>
<tbody>
<tr>
<td>4000-4003</td>
<td>list(1)</td>
</tr>
<tr>
<td>4004-4007</td>
<td>list(2)</td>
</tr>
<tr>
<td>4008-4011</td>
<td>list(3)</td>
</tr>
<tr>
<td>4012-4015</td>
<td>list(4)</td>
</tr>
<tr>
<td>4016-4019</td>
<td>list(5)</td>
</tr>
<tr>
<td>4020-4023</td>
<td>i</td>
</tr>
</tbody>
</table>

Table 26.2: Memory Map of list and i

When setting list(6), it computes the address 4000 + 4 * (6-1) = 4020, which is the address of the integer variable 'i'. It places a 4 byte integer 0 there, overwriting 'i', and causing the loop to start over.

This demonstrates the value of using named constants or variables to specify array boundaries!

integer :: list(LIST_MAX), i
do i = 1, LIST_MAX
    list(i) = 0
    print *, i
endo

It's easy to type 6 when you meant 5, but not so easy to type LIST_MAX+1 when you meant LIST_MAX.

26.11.3 Pinpointing Crashes with a Debugger

Out-of-bounds array subscripts can cause a variety of problems.

- They may cause data corruption that reveals itself immediately as shown above.
- They may corrupt other data in a way that doesn’t cause any problems, or causes problems at a later stage of the program.
- They may cause different types of corruption depending on the compiler, the operating system, and the types of optimizations you compiled with. (You will often hear inexperienced programmers claim that the optimizer broke their program. This is possible in theory, but the vast majority of the time, the optimizer simply exposed a bug in their code by changing the type of corruption it caused.)
• Finally, if you’re lucky, a runaway subscript may cause your program to crash. In Unix, this is usually associated with a 
  segmentation fault or a bus error. Both of these errors indicate that the program attempted an illegal memory access.

  A segmentation fault means an attempt to access memory in a way that’s not allowed (e.g. a memory address not allocated for 
data), whereas a bus error indicates an attempt to access memory that does not physically exist, an attempt to write to a ROM, 
etc.

  These errors often cause the program to dump a core, which is a snapshot of the program code and data as it appeared in 
  memory at the moment of the crash. The core file can be used by a debugger to pinpoint exactly which statement in the 
  program was executing when the crash occurred. For most Unix compilers, compiling with the -g flag causes more useful 
  information for debuggers to be placed in the executable file. Increasing the optimization level (-O, -O2, -O3) reduces the 
  debugger’s ability to pinpoint problems. For more information, consult the documentation on your compiler and debugger.

26.12 Fortran 90 Vector Operations

26.12.1 Implicit Loops

Fortran provides shorthand notation for many simple array operations:

```fortran
list(1:list_size) = 0.0d0
```

is the same as

```fortran
do i = 1, list_size
    list(i) = 0.0d0
enddo
```

The `list(1:list_size)` is an example of an implicit loop for operating on the array. The latter example is an explicit loop. 
There is no difference in performance between these two Fortran code segments. The compiler will translate both to the same 
sequence of machine instructions.

Caution MATLAB has similar notations, but in MATLAB, the implicit loop is many times faster, because MATLAB is an 
interpreted language. The implicit loop in MATLAB would use a compiled loop, which is roughly equivalent to the loop 
in Fortran. The explicit loop in MATLAB is interpreted, hence explicit loops should be avoided at all cost in MATLAB.

The general form of an implicit loop of this type is

```fortran
array(start:end:stride)
```

Examples:

```fortran
vector(1:vector_length) = vector(1:vector_length) + 4.0d0
```

same as

```fortran
do i = 1, vector_length
    vector(i) = vector(i) + 4.0d0
enddo
```

Example:

```fortran
vector1(1:vector_length) = vector2(1:vector_length)
```

same as
do i = 1, vector_length
    vector1(i) = vector2(i)
enddo

Example:

vector(1:vector_length:2) = 3.0d0 ! Odd-numbered subscripts

same as

do i = 1, vector_length, 2
    vector(i) = 3.0d0
enddo

Example:

print *, vector1(1:vector_length)

same as

print *, (vector1(i), i=1,5)

It is legal, but incompetent to omit the starting and ending subscripts from an implicit loop:

vector1 = vector2 + vector3

If you do this, the program will use the minimum and maximum subscripts for the arrays. This is fine if all elements contain useful data. If the array is not fully utilized, however, the loop above will add all the useful elements and all the garbage. For example, if the arrays hold 1,000,000 elements, and the vectors they contain use only 3, then your loop will take about 333,333 times as long as it should.

26.12.2 Where

where (a > 0)
    b = sqrt(a)
elsewhere
    b = 0.0
endwhere

Same as

do i = 1, LIST_SIZE
    if ( a(i) > 0 ) then
        b(i) = sqrt(a(i))
    else
        b(i) = 0.0
    endif
enddo

26.13 Code Quality

1. Use named constants for the size of all arrays
2. Use size_t.
26.14 The Memory Hierarchy

26.14.1 Virtual Memory

Average disk access time for a random block is a few milliseconds, write buffering, disk caching and predictive reads reduce this.

26.14.2 Cache

Multiple levels, hit ratio.

26.14.3 Memory Access Time

Example: 30ns DRAM, 2ns cache. Hit ratio 0.9, avg access time is 0.9 * 2 + 0.1 * 30 = 4.8 ns. Hit ration 0.5, avg access time is 0.5 * 2 + 0.5 * 30 = 16 ns.

With 1% VM swapping: 4ms disk (1ms avg access time), 30ns DRAM, 2ns cache. 0.01 * 1,000,000ns + 0.69 * 30ns + 0.3 * 2ns = 10,021.3ns

26.15 Performance

1. Eliminate arrays whenever possible Example: Input data into an array and then compute average
2. Traverse arrays sequentially rather than jump around in order to maximize memory performance
3. Compilers and optimizers assume that arrays are traversed in order.

26.15.1 Code Examples

26.16 Self-test

1. When should arrays be used? Why?
2. Why is it important to use dynamic memory allocation for all large arrays?
3. How does a lookup table improve program performance? What is the down side of lookup tables?
4. What is an out-of-bounds error? What kinds of problems can it cause?
5. When passing an array as an argument to a subprogram, what must always be passed with it? Why?
Chapter 27

Strings

27.1 Motivation

Most programs need to communicate with end users. To facilitate this, we have character sets such as ASCII, ISO-Latin1, and Unicode. A sequence of such characters is called a string.

All input from and output to a terminal is in the form of a strings, even if that input and output is numeric. When you type in the number 451.32 as input to a program, you are sending the program the characters ‘4’, ‘5’, ‘1’, ‘.’, ‘3’, and ‘2’. The Fortran read subroutine then converts this string of characters to the appropriate binary format (usually two’s complement or IEEE floating point) and stores it in some variable.

Some programs are meant to process character data, not numeric data. For example, a program to manipulate genetic data might use strings of ’a’ for adenine, ’c’ for cytosine, ’g’ for guanine, ’t’ for thymine (and perhaps ’u’ for uracil, which is almost identical to thymine).

27.2 Defining String Variables

Strings are difficult for compilers and interpreters to deal with efficiently, because they vary so much in length. Conserving memory when dealing with strings requires frequently allocating and deallocating small blocks of memory, which is costly in terms of CPU time. The only way to avoid this is by over-allocating strings (e.g. allocate 100 characters for a string even if we only need 12 at the moment). Hence, there is a trade-off to be made between CPU efficiency and memory efficiency.

There are programming techniques to minimize the cost of string manipulation, and thus improve the efficiency of programs that process character data. Some of them will be mentioned in the sections that follow.

A string in Fortran is, for all practical purposes, an array of characters. However, Fortran gives special treatment to strings, so the syntax for working with them is slightly different than it is for arrays of numbers or logical values.

When defining a string, we must indicate the maximum length. The syntax for this changed starting with Fortran 90, but the old syntax used through Fortran 77 is still supported for backward compatibility. All new code should use the Fortran 90 syntax, but older programs using the Fortran 77 syntax will compile with a Fortran 90 compiler.

The Fortran 90 syntax for defining string variables is:

```fortran
character(maximum-length) :: name1, name2, ...
```

As always, we should use named constants to define things like the maximum length of a string.

```fortran
module constants
    integer, parameter :: MAX_NAME_LENGTH=40
end module

program string_example
```


implicit none

character(MAX_NAME_LEN) :: name

print *, 'Please enter your name:
read *, name
print *, 'Hello, ', name, '!
end program

The Fortran 77 and earlier syntax for defining string variables is:

character*maximum-length :: name1, name2, ...

This syntax should not be used for new programs. It is presented here so that you will recognize it if you see it in older code.

27.3 String Constants

A string constant, as you have already seen, is a sequence of characters between quotation marks. You can use either single or double quotes. Single quotes are most common, since they do not require pressing the shift key. However, if a string must contain a single quote, then it must be delimited using double quotes. If a string starts with a `, then the compiler will think the next ` is the end of the string.

print *, "You can't delimit this string with "'

27.4 Truncation and Padding

String variables are rarely "full", i.e. they rarely contain a string of the maximum length. Fortran pads string variables with spaces to fill the unused elements in the string. Hence, the following two statements are equivalent:

name = 'Alfred E. Neumann'
name = 'Alfred E. Neumann '

Caution If we attempt to assign a string that is too large for the variable, it will be truncated. This can be disastrous for some programs!

module constants
  integer, parameter :: MAX_NAME_LENGTH=10
end module

program string_example
  implicit none

  character(MAX_NAME_LEN) :: name

  name = 'Alfred Hitchcock'
  print *, name
end program

Output:
Alfred Hit
27.5 Common String Operations

Fortran supports many operators and intrinsic functions for manipulating strings.

27.5.1 Concatenation

Strings can be concatenated (pasted end-to-end) using the // operator:

```fortran
module constants
  integer, parameter :: MAX_NAME_LENGTH=20
end module

program string_example
  ! Variable definitions
  character(MAX_NAME_LEN) :: first_name, last_name
  character(MAX_NAME_LEN * 2) :: full_name
  ! Statements
  first_name = "Alfred"
  last_name = "Neumann"
  full_name = first_name // last_name
  print *, full_name
end program
```

Output?

27.5.2 Trimming

One of the most common operations with strings is trimming, or removing the trailing spaces to get just the "meat". Trimming is done with the trim() intrinsic function. Note that we cannot change the maximum size of a string variable. The trim() function allocates a smaller temporary string, whose maximum length is the actual length of its argument.

```fortran
character(20) :: name
name = 'Alfred Nobel'
print *, name, ' is the namesake for the Nobel prize.'
print *, trim(name), ' is the namesake for the Nobel prize.'
```

Alfred Nobel is the namesake of the Nobel prize.
Alfred Nobel is the namesake of the Nobel prize.

We can use trimming to fix the issue with concatenating names above.

```fortran
module constants
  integer, parameter :: MAX_NAME_LENGTH=20
end module

program string_example
  ! Variable definitions
  character(MAX_NAME_LEN) :: first_name, last_name
  character(MAX_NAME_LEN * 2) :: full_name
  ! Statements
  first_name = "Alfred"
  last_name = "Neumann"
  full_name = trim(first_name) // ' ' // trim(last_name)
  print *, full_name
end program
```
27.5.3 String length

Sometimes we want to know the length of a string. The `len()` and `len_trim()` functions return an integer length of the string variable and the actual content, respectively.

```fortran
character(20) :: name
name = 'Alfred Nobel'
print *, len(name)
print *, trim_len(name)
```

```
20
12
```

`trim_len(string)` is equivalent to `len(trim(string))`

27.5.4 Substrings

Sometimes we want to extract part of a string. We can simply specify a starting and ending subscript separated by a `':'`. If either is omitted, and a `':'` is present, the missing subscript is assumed to be `1` or the maximum subscript for the string.

This syntax can also be used on the left-hand-side of an assignment statement.

```fortran
character(MAX_NAME_LEN) :: name
name = 'Alfred Pennyworth'
print *, name(1:6) ! Alfred
print *, name(8:17) ! Pennyworth
print *, name(:6) ! Alfred
print *, name(8:) ! Pennyworth
name(8:) = 'E. Neumann" ! name is now 'Alfred E. Neumann'
```

27.6 Strings as Subprogram Arguments

When a subprogram takes a string as an argument, we may want to pass strings of different lengths.

```fortran
module constants
    integer, parameter :: MAX_CHROMOSOME_LEN = 1000000, &
                      MAX_GENE_LEN = 20000, &
                      MAX_MARKER_LEN = 20
end module

program string_subprograms
    implicit none
    character(MAX_CHROMOSOME_LEN) :: chrom1, chrom2
    character(MAX_GENE_LEN) :: eye_gene1, eye_gene2, hair_gene1
    character(MAX_MARKER_LEN) :: start_seq, stop_seq
    integer :: start_offset, code_offset, stop_offset
    ...
    ! Find location of sequences within a chromosome
    start_offset = locate(chrom1, start_seq)
    stop_offset = locate(chrom1, stop_seq)
```
code_offset = locate(chrom1, gene_seq)

... end program

Unlike other arrays, it is not necessary to pass the maximum length of a string. We simply define dummy string variables with a * for the size, and the compiler takes care of the rest. The compiler will pass the length of the string to the subprogram for us, and we can retrieve it using len(string), where string is the name of the dummy argument.

subroutine sub(string)
  character(*) :: string

  ! Produces the correct length of the string passed in each time
  print *, len(string)
end subroutine

The locate function above can then be implemented as follows:

function locate(big_sequence, sub_sequence)
  integer :: locate
  character(*) :: big_sequence, sub_sequence

  integer :: sub_len
  character :: trimmed_seq

  ! Indicate that sub_sequence was not found
  locate = -1

  ! Set locate to index of first match, if found
  trimmed_seq = trim(sub_sequence)
  sub_len = len(trimmed_seq)
  do i = 1, trim_len(big_sequence)
    if ( big_sequence(i:i+sub_len) == trimmed_seq ) then
      locate = i
      exit
    endif
  enddo
end function

27.7 Command-line Arguments

Knowing how to handle string variables allows us to process arguments to a program from the command line. You’ve used programs that take arguments on the command line:

mypc: ape asg01.f90
mypc: f90 asg01.f90

How do programs like ape and f90 get arguments like asg01.f90 from the command line?

In C, command-line arguments are received in the argument variable argv in the main program.

cla.c.dbk

In a Fortran program, this is done using the intrinsic subroutine getarg().

Suppose we want to write a program that computes base\textsuperscript{exponent}, but instead of asking the user to input the base and exponent, it takes them on the command line.
The getarg() subroutine grabs a command line argument and stores it in a string variable. The first argument after the program name is argument 1, and so on. Argument 0 is the name of the program as it was invoked from the Unix command-line.

```fortran
character(MAX_INTEGER_DIGITS) :: program_name, &
   base_string, exponent_string

call getarg(0, program_name)
call getarg(1, base_string)
call getarg(2, exponent_string)
```

If the argument number given is greater than the actual number of arguments, getarg() simply returns a blank string. We can use this to check whether the correct number of arguments were given. Note that argument 1 will never be missing if argument 2 is present, so we need only check the last argument to ensure that all are there.

```fortran
if (exponent_string == '') then
   print *, trim(program_name), ': Usage: power base exponent'
   stop
endif
```

To convert the strings to integers or reals, we use a read statement, with the string variable as the unit:

```fortran
integer :: base, exponent
read (base_string, *) base
read (exponent_string, *) exponent
```

---

```
! Program description:
! Compute a power of command line arguments base and exponent
!
! Arguments:
! First: An integer base
! Second: An integer exponent

module constants
   ! Global Constants
   integer, parameter :: MAX_INTEGER_DIGITS = 10
end module constants

! Main program body
program power
   use constants         ! Constants defined above
   implicit none
   ! Disable implicit declarations (i-n rule)
   implicit none
   ! Variable definitions
   integer :: base, exponent
   character(MAX_INTEGER_DIGITS) :: base_string, exponent_string
   call getarg(0, program_name)
call getarg(1, base_string)
call getarg(2, exponent_string)
if (exponent_string == '') then
   print *, trim(program_name), ': Usage: power base exponent'
   stop
endif
integer :: base, exponent
read (base_string, *) base
read (exponent_string, *) exponent
```

```fortran
  print *, base, exponent
  print *, base ** exponent
```

```
! Modification history:
! Date   Name  Modification
! 2011-03-25 Jason Bacon Begin
```

---
! First command line argument is the base, second is the exponent
call getarg(1, base_string)
call getarg(2, exponent_string)

! Make sure user provided both base and exponent
if ( exponent_string == '' ) then
  stop 'Usage: power base exponent'
endif

! Convert strings to integers
read (base_string, *) base
read (exponent_string, *) exponent

! Compute power
print *, base, ' ** ', exponent, ' = ', base ** exponent
end program

If we do not know how many command line arguments there are, we can use the fact that getarg() returns a blank string for any index higher than the number of arguments.

### 27.8 Code Quality

1. Use named constant + 1 for size of all string arrays
2. Do not use dangerous library functions strcpy(), strcat(), sprintf(), etc.
3. Use size_t

### 27.9 Performance

1. Avoid copying strings. Use pointers instead.
2. Avoid repeated use of strlen().

#### 27.9.1 Code Examples

### 27.10 Self-test

1. Write a Fortran program that simply prints all of the command line arguments.
Chapter 28

File I/O

28.1 Motivation

Files are used for two main reasons:

- Long term storage: To store data while the programs that use it are not running.
- Short-term storage: To store temporary data that is too big for an, or is not accessed often enough to warrant keeping it in an array.

28.2 Filesystem Structure

As you saw in Chapter 7, files are organized into a tree-shaped structure of directories, also known as folders.

All the information presented there regarding the current working directory, relative and absolute pathnames, also applies within Fortran programs. An absolute or relative pathname is represented in Fortran as a string constant, variable, or expression.

28.3 C File Streams

28.3.1 fopen() and fclose()

28.3.2 Stream Read Functions

28.3.3 Stream Write Functions

28.3.4 Example Program

/***************************************************************************/
* Usage: sort input-file output-file
* Sort a list of real numbers in input-file, placing the results
* in output-file. Both input and output files contain one
* number per line.
* Arguments:
* input-file: file to be sorted
* output-file: file to receive the sorted list
/***************************************************************************/
#include <stdio.h>
#include <stdlib.h>
#include <sysexits.h>

double *read_list(size_t *size, const char filename[]);
int print_list(const double list[], size_t list_size, const char filename[]);
void sort_list(double list[], size_t list_size);
void usage(void);

int main(int argc, char *argv[])
{

double *list;
size_t list_size;
int status = EX_OK;
char *input_file, // Just for readability
     *output_file;

if ( argc != 3 )
    usage();
input_file = argv[1];
output_file = argv[2];

list = read_list(&list_size, input_file);
if ( list == NULL )
{
    perror(input_file);
    return EX_NOINPUT;
}
sort_list(list, list_size);
if ( print_list(list, list_size, output_file) != EX_OK )
{
    perror(output_file);
    status = EX_CANTCREAT;
}
free(list);
return status;
}
if ( input_file != NULL )
{
    fscanf(input_file, "%zu", size_ptr);  // No & here, since size_ptr is a pointer
    list = (double *)malloc(*size_ptr * sizeof(*list));
    for (c = 0; c < *size_ptr; ++c)
        fscanf(input_file, "%lf", &list[c]);
    fclose(input_file);
    return list;
}

/*
 * Print contents of a list to a file.
 */
int print_list(const double list[], size_t list_size, const char filename[])
{
    size_t c;
    FILE *output_file;
    
    /*
    * If open fails, just return an error code to the caller. This is a
    * general-purpose function, usable in many programs, so it should not
    * take any specific action.
    */
    output_file = fopen(filename, "w");
    if ( output_file != NULL )
    {
        for (c = 0; c < list_size; ++c)
            fprintf(output_file, "%f\n", list[c]);
        return EX_OK;
    }
    fclose(output_file);
    return EX_CANTCREAT;
}

/*
 * Sort list using selection sort algorithm.
 */
void sort_list(double list[], size_t list_size)
{
    size_t start,
            low,
            c;
    double temp;

    for (start = 0; start < list_size - 1; ++start)
    {
        /* Find lowest element */
        low = start;
        for (c = start + 1; c < list_size; ++c)
            if ( list[c] < list[low] )
                low = c;

        /* Swap first and lowest */
        temp = list[start];
        list[start] = list[low];
        list[low] = temp;
void usage(void)
{
    fputs("Usage: selsort-files input-file output-file\n", stderr);
    exit(EX_USAGE);
}

28.4 C Low-level I/O

28.4.1 open() and close()

28.4.2 read()

28.4.3 write()

28.5 Fortran File Operations

Fortran, like most programming languages, uses the refrigerator model for file access. Like a refrigerator, a file must be opened before we can put anything into it (write) or take anything out (read). After we’re done reading or writing, it must be closed.

A file is essentially a sequence of bytes stored on a disk or other non-volatile storage device.

Access to files is, for the most part, sequential, meaning we start reading a file at the first byte, and go forward from there. After we read the first byte, the next read operation automatically gets the second, and so on. It is possible to control the order in which we read or write data to a file, but doing non-sequential access is a bit more cumbersome than it is with an array. Unlike array access, it takes separate statements to go to a different location within the file and then read or write data.

28.5.1 Open and Close

Before we can access a file from Fortran, we must open it. The open statement creates a new unit, which can then be used in read and write statements to input from or output to the file.

As you saw in Section 21.2, all input and output in Fortran uses unit numbers.

The unit number is Fortran’s way of representing the more general concept of a file handle. The term “file handle” is a metaphor for a mechanism used to grasp or control something. Just like a suitcase or door has a handle that allows you to manipulate it, files in Fortran and other languages have conceptual "handles" for manipulating them.

You might be wondering, isn’t this what the filename is for? To some extent, yes, but when we’re reading or writing a file, we need more than just the name. We need to keep track of where we are in the file, for example.

The open statement creates a structure containing information about the file, such as its name, whether we’re reading or writing to it, where we are in the file at a given moment, etc. As a Fortran programmer, you do not need to keep track of this information yourself. The compiler and operating system take care of all of this for you. All you need in order to work with an open file is the unit number. This unit number is your handle to the file.

It is your job as a Fortran programmer to choose a unique unit number for each file you open. Low-numbered unit numbers such as 0, 1, and 2 are reserved for special units like INPUT_UNIT, OUTPUT_UNIT, and ERROR_UNIT. Most Fortran programmers use 10 as the lowest unit number, and go up from there if they need more than one file open at a time.
module constants
integer, parameter :: &
   MAX_PATH_LEN = 1024, &
   CHROMOSOME_UNIT = 10
end module

program files
implicit none
integer open_status, close_status
character(MAX_PATH_LEN) :: filename

filename = 'input.txt'
open (unit=CHROMOSOME_UNIT, file=filename, status='old', &
   iostat=open_status, action='read', position='rewind')
if ( open_status /= 0 ) then
   print *, 'Could not open ',filename,' for reading.', &
   'unit = ', unit
   stop
endif
...

close(CHROMOSOME_UNIT, iostat=close_status)
if ( close_status /= 0 ) then
   print *, 'Error: Attempt to close a file that is not open.', &
   'unit = ', CHROMOSOME_UNIT
   stop
endif
end program

The open statement uses a number of tags to specify which file to open and how it will be used.

Required tags:

- **unit**: integer unit number to be used by read, write, and close.
- **filename**: String constant, variable, or expression representing the absolute or relative pathname of the file.
- **status**: String constant, variable, or expression that reduces to:
  - ‘old’: For existing files, usually used when reading.
  - ‘new’: Used when writing to a file that does not yet exist.
  - ‘replace’: Used to overwrite a file that already exists.
- **iostat**: Integer variable to receive the status of the open operation. If the file is opened successfully, the variable is set to 0. Otherwise, it will contain a non-zero error code that indicates why the file could not be opened. (Does not exist, no permission, etc.)

Optional tags:

- **action**: String
  - ‘read’: Open for reading only
  - ‘write’: Open for writing only
  - ‘readwrite’: Allow both reading and writing
- **position**: String
– 'rewind': Start at beginning of file
– 'append': Writes add to file rather than overwrite

The close statement makes sure any writes to a file opened for writing are complete, and then disables the unit. About the only way a close can fail is if the unit does not represent an open file. This generally means there’s a bug in the program, since a close statement should only be attempted if the open succeeded.

### 28.5.2 Read

The read statement works for files exactly as it does for the standard input. Recall that you can actually make the standard input refer to a file instead of the keyboard by using redirection in the Unix shell:

```
shell> asg02 < input.txt
```

The above example reads from standard input using a statement such as `read (*,*) variable`. Recall that the first `'*'` represents the default unit (standard input) and the second represents the default format.

When reading from a file that was opened by the program, we simply replace the first `'*'` with an explicit unit number.

We also introduce here the use of the iostat tag. The iostat variable will receive 0 if the read is successful, and a non-zero error code if it failed (at end of file, file is not open, etc.) Technically, the iostat tag could and should be used when reading from the standard input as well, but it was omitted in Chapter 21 for simplicity.

```fortran
integer :: read_status
character(MAX_CHROMOSOME_LEN) :: chromosome1
read (CHROMOSOME_UNIT, *, iostat=read_status) chromosome1
if ( read_status /= 0 ) then
  print *, 'Error reading file, unit = ', CHROMOSOME_UNIT
  stop
endif
```

### 28.5.3 Write

The same ideas apply to write as to read.

```fortran
integer :: write_status
character(MAX_CHROMOSOME_LEN) :: chromosome1
write (CHROMOSOME_UNIT, *, iostat=write_status) chromosome1
if ( write_status /= 0 ) then
  print *, 'Error writing file, unit = ', CHROMOSOME_UNIT
  stop
endif
```

### 28.6 File Format Standards

Netcdf, hdf5, nifti, minc
28.7 Code Quality

28.8 Performance

28.8.1 Code Examples

28.9 Self-test

1. What are the general steps involved in accessing a file from within a Fortran program?

2. How does reading and writing files differ from reading and writing to/from the terminal (keyboard and screen)?
Chapter 29

Matrices

Assigned readings: Sec 9.1, 9.2

29.1 Motivation

We have seen how to use arrays to store one-dimensional lists in memory. Often there is a need to store and manage data which is conceptually multidimensional.

29.1.1 Tables

A table is any two-dimensional arrangement of data. Common uses are timetables, score tables, etc.

<table>
<thead>
<tr>
<th></th>
<th>Sun</th>
<th>Mon</th>
<th>Tue</th>
<th>Wed</th>
<th>Thu</th>
<th>Fri</th>
</tr>
</thead>
<tbody>
<tr>
<td>8:00am</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>9:00</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>10:00</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>11:00</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 29.1: Time Table

29.1.2 Grids

Grids are representations of values at physical locations in two dimensions. Grids are commonly used to represent weather data.

<table>
<thead>
<tr>
<th>Lat/Long</th>
<th>86</th>
<th>87</th>
<th>88</th>
<th>89</th>
<th>90</th>
<th>91</th>
</tr>
</thead>
<tbody>
<tr>
<td>44</td>
<td>56</td>
<td>54</td>
<td>53</td>
<td>55</td>
<td>59</td>
<td>52</td>
</tr>
<tr>
<td>43</td>
<td>57</td>
<td>56</td>
<td>58</td>
<td>60</td>
<td>55</td>
<td>54</td>
</tr>
<tr>
<td>42</td>
<td>59</td>
<td>61</td>
<td>58</td>
<td>55</td>
<td>56</td>
<td>56</td>
</tr>
</tbody>
</table>

Table 29.2: Temperature

29.1.3 Bricks

A brick is like a grid, but with more than two dimensions. Bricks of data can represent things like MRI images, or weather data including altitude as well as latitude and longitude.
29.1.4 Systems of Equations

Systems of polynomial equations are commonly stored as a matrix of coefficients, which can then be solved using techniques such as Gaussian elimination.

29.1.5 Matrix Representation

In all of the examples above, we wish to identify one piece of data in a collection using multiple coordinates. For example, we may want to check our schedule for (Mon, 10:00), find the temperature at latitude, longitude (43°, 78°), or measure brain activity at 10mm sagittal, 8mm coronal, and 12mm transverse (10, 8, 12).

Multidimensional data can be represented as a matrix, which is a multidimensional collection of numbers. A vector, defined in Chapter 26, is simply a matrix where only one dimension has a measurement greater than 1. For a two-dimensional matrix, the row subscript is first by convention.

\[
\begin{array}{ccc}
    a_{1,1} & a_{1,2} & a_{1,3} \\
    a_{2,1} & a_{2,2} & a_{2,3} \\
    a_{3,1} & a_{3,2} & a_{3,3} \\
\end{array}
\]

Table 29.3: Generic 2D Matrix

29.2 Multidimensional Arrays

C does not technically support multidimensional arrays. However, we can define an array or arrays, which accomplished the same thing.

```c
#define MAX_ROWS 100
#define MAX_COLS 100

double coefficients[MAX_ROWS][MAX_COLS];
```

Statically sized 1-dimensional arrays are wasteful enough, but the waste increases by an order of magnitude with each dimension we add, so this is generally a bad practice unless the maximum matrix size is small.

The key to conserving memory with matrices is hidden in plain view in the argv[] array received by main().

```c
int main(int argc, char *argv[])
```

The argv[] array is essentially a 2-dimensional array of characters. We can use the same concept for any 2-dimensional array.

TBD: Graphic of a pointer array

To implement a pointer array, we first allocate a 1-dimensional array of pointers to the data type contained in the matrix.

Each of these pointers will point to a 1-dimensional array of values representing a row in the matrix.

```c
/***************************************************************************
* Description: *
* Read a matrix into a pointer array and print it out to verify. *
* *
* History: *
* Date Name Modification *
* 2017-08-25 Jason Bacon Begin *
***************************************************************************/

#include <stdio.h>
#include <sysexits.h>
#include <stdlib.h>
```
typedef double real_t;

int main(int argc, char *argv[])
{
    size_t rows, cols, r, c;
    real_t **matrix;

    printf("Rows? ");
    scanf("%zu", &rows);
    printf("Cols? ");
    scanf("%zu", &cols);

    /* Allocate pointer array */
    matrix = (real_t **)malloc(rows * sizeof(real_t *));

    /* Read matrix data separated by whitespace. Well-formatted input
     * will have a newline after each row.
     */
    for (r = 0; r < rows; ++r)
    {
        matrix[r] = (real_t *)malloc(cols * sizeof(real_t));
        for (c = 0; c < cols; ++c)
            scanf("%lf", &matrix[r][c]);
    }

    /* Print matrix one row per line. */
    for (r = 0; r < rows; ++r)
    {
        for (c = 0; c < cols; ++c)
            printf("%5.2f", matrix[r][c]);
        putchar('n');
    }

    /* Free the arrays */
    for (r = 0; r < rows; ++r)
        free(matrix[r]);
    free(matrix);
    return EX_OK;
}

2 3
2.0 4.3 1.0
9.7 4.7 8.2

In Fortran, we store matrix data in a multidimensional array. Defining multidimensional arrays is done much like one-dimensional
arrays. We add more dimensions separated by commas:

    double precision, allocatable :: coefficients(:, :)  
    integer :: rows, cols

Note that as the number of dimensions increases, so does the potential for memory waste if the array size is larger than needed. If
we use 50 elements in an array of 100, half the array is wasted. If we store a 50x50 matrix in a 100x100 array, 3/4 of the memory
(7,500 elements) is wasted. If we store a 50x50x50 brick in a 100x100x100 array, 7/8 of the memory (875,000 elements) is
wasted. Therefore, using dynamic memory allocation becomes increasingly important as the number if dimensions grows.
read *, rows, cols

allocate(matrix(1:rows, 1:cols), stat=allocate_status)

if ( allocate_status /= 0 ) then
   print *, 'Error: Unable to allocate ', rows, ' x ', cols, ' matrix.'
   stop
endif

Working with multidimensional arrays will always involve a nested loop, i.e. a loop inside a loop. In Fortran, the outer loop should traverse the columns, and the inner loop should traverse the rows. The reason for this is explained in Section 29.4.

29.3 Reading and Writing Matrices

When matrix data is stored in a file, it is most convenient to store each row of the matrix in on line of the file. In addition, it is helpful, but not necessary, to store the dimensions of the matrix at the beginning of the file. This makes it easy to read the matrix. The example below shows a matrix with two rows and three columns as it would appear in the input stream. Note that it makes no difference whether it is read from the keyboard or a file. This is how the data would appear on the screen as it is typed, or in a text file opened with an editor.

<table>
<thead>
<tr>
<th>2 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>10.4 5.8 -1.2</td>
</tr>
<tr>
<td>0.6 3.9 8.3</td>
</tr>
</tbody>
</table>

With the file in this format, we can easily read the data into a two-dimensional array, remembering the "one statement equals one line" rule for reading input. That is, each line of the file must be read using a single read statement.

!-----------------------------------------------------------------------
! Program description:
! Input a matrix from standard input and echo to standard output
!-----------------------------------------------------------------------

!-----------------------------------------------------------------------
! Modification history:
! Date   Name          Modification
! 2011-03-26 Jason Bacon Begin
!-----------------------------------------------------------------------

! Main program body
program echo_matrix
   ! Disable implicit declarations (i-n rule)
   implicit none

   ! Variable definitions
double precision, allocatable :: matrix(:,:)
integer :: rows, cols, r, allocate_status, read_status

   ! Get size of matrix
   read *, rows, cols

   ! Allocate array to hold matrix
   allocate(matrix(1:rows, 1:cols), stat=allocate_status)
   if ( allocate_status /= 0 ) then
      print *, 'Error allocating ', rows, ' x ', cols, ' matrix.'
      stop
   endif
29.4 Importance of Traversal Order

Fortran is a column-major language, which means that all the elements of a column in a two-dimensional matrix are adjacent in memory.

```fortran
double precision :: matrix(1:2, 1:3)

Memory map:

<table>
<thead>
<tr>
<th>Address</th>
<th>Array Element</th>
</tr>
</thead>
<tbody>
<tr>
<td>2000</td>
<td>matrix(1,1)</td>
</tr>
<tr>
<td>2008</td>
<td>matrix(2,1)</td>
</tr>
<tr>
<td>2016</td>
<td>matrix(1,2)</td>
</tr>
<tr>
<td>2024</td>
<td>matrix(2,2)</td>
</tr>
<tr>
<td>2032</td>
<td>matrix(1,3)</td>
</tr>
<tr>
<td>2040</td>
<td>matrix(2,3)</td>
</tr>
</tbody>
</table>
```

As discussed in Chapter 18, most modern computers use virtual memory, which extends the apparent size of RAM using swap space on disk.

When a program accesses a given memory address, it is really using a virtual address, and the data stored at that virtual address could be in RAM or in swap space. If the data is in RAM, it will take nanoseconds to access. If it is in swap, it will take milliseconds, about 1,000,000 times longer.

Because it is more efficient to access disk in large chunks than in individual bytes, virtual memory is divided into pages. If a particular virtual address is in RAM, then the entire page surrounding it is in RAM, and if it is in swap, the entire page is in swap.

The goal when writing code is to minimize the frequency of accessing a different page than the last one accessed. If we access a value that is in swap, then we have to go to disk, which is expensive. At this point, the system moves the entire page into RAM, not just the value. This is called a page fault. If the next memory access is in the same page, there is no chance of causing a page fault.

Every time a page fault occurs, your program has to wait for the slow disk access, and loses a lot of time:

```
<table>
<thead>
<tr>
<th>Time</th>
<th>Instructions</th>
</tr>
</thead>
<tbody>
<tr>
<td>4 ms</td>
<td>3 billion</td>
</tr>
</tbody>
</table>
```

Hence, if we structure our programs so that they do as many consecutive memory references as possible within the same page, we will reduce the number of page faults, and increase program performance.

This is where it is useful to know that Fortran is a column-major language. Since each column of a two-dimensional array is grouped together in memory, much or all of the column will be in the same page. Therefore, if our program works within a column for as long as possible, instead of jumping to a different column every time, it may run faster.
Example 29.1 Comparison of Row-major and Col-major Access

PC with 2G RAM, initialize a matrix of nearly 2G. (Requires a small amount of swap to be used, since OS and other programs take some RAM.)

The code below traverses the matrix in column-major order, i.e. is completely sweeps one column before going to the next. This program took an average of 5.7 seconds to run over several trials.

```fortran
  double precision :: matrix(:, :)

  ! Matrix of almost 2Gbytes
  rows = 15000
  cols = 15000

  ! Allocate
  allocate(matrix(1:rows, 1:cols))

  # Column-major access
  do c=1, cols
    do r=1, rows
      matrix(r,c) = 0.0d0
    enddo
  enddo
```

If we switch the inner and outer loops, the program will traverse the matrix in row-major order, i.e. it will sweep each row before going to a new one, which means it accesses a different column (and hence a different page) every time it accesses the array. This slight change to the code caused the program to take an average of 41 seconds to run instead of 5.7!

```fortran
  # Row-major access
  do r=1, rows
    do c=1, cols
      matrix(r,c) = 0.0d0
    enddo
  enddo
```

Note that when reading or writing a matrix, we don’t have much choice but to access it in row-major order. This doesn’t make that much difference, since I/O is slow anyway. Traversal order is most important when processing a matrix entirely in memory. (Matrix addition, multiplication, Gauss elimination, etc.)

29.5 Multidimensional Arrays as Arguments

When passing a multidimensional array as an argument to a subprogram, is is even more critical that the dimensions are known. With a one-dimensional array, the subprogram must know the size of the array in order to avoid out-of-bounds errors.

With a multidimensional array, we still have the possibility of out-of-bounds errors. In addition, if the correct dimensions are not known to the subprogram, it will not be able to compute the correct location of an element from the subscripts! Look again at the memory map of a small 2D array:

```fortran
  double precision :: matrix(2, 3)

  Memory map:
  2000  matrix(1,1)
  2008  matrix(2,1)
  2016  matrix(1,2)
  2024  matrix(2,2)
  2032  matrix(1,3)
  2040  matrix(2,3)
```
The memory address of an element is computed as:

\[
\text{address}(r, c) = \text{base} + (\text{total-rows} \times (c-1) + (r-1)) \times \text{sizeof(type)}
\]

For an array of double precision values based at address 2000:

\[
\text{address}(r, c) = 2000 + (2 \times (c-1) + (r-1)) \times 8
\]

\[
\text{address}(2, 3) = 2000 + (2 \times 2 + 1) \times 8 = 2040
\]

Since the calculation of an address involves the total number of rows in the matrix, we must at least get this right just to be able to access the array elements correctly. Getting the number of columns right is also essential to prevent out-of-bounds problems.

```fortran
subroutine print_matrix(matrix, rows, cols)
  ! Disable implicit declarations (i-n rule)
  implicit none

  ! Dummy variables
  integer, intent(in) :: rows, cols
  double precision, intent(in) :: matrix(rows, cols)

  ! Local variables
  integer :: r

  do r = 1, rows
    print *, matrix(r, 1:cols)
  enddo
end subroutine
```

### 29.6 Fortran Intrinsic Functions

Fortran has many intrinsic functions for performing common matrix operations such as multiplication, transposition, etc. The functions provided are well optimized, and should be used in real-world applications.

However, students new to Fortran are encouraged to write their own functions for matrix transpose, multiplication, etc. in order to gain a better understanding of programming. This will help you understand and appreciate the complexity, advantages, and limitations of intrinsic functions and the Fortran language.

### 29.7 Homework

1. What does column-major mean?
2. How should two-dimensional arrays be traversed in Fortran? Why?
3. Why do two-dimensional arrays passed as arguments have to have the correct dimensions for the subprogram?
Chapter 30

Software Performance

30.1 Motivation

Many computing tasks, especially in scientific research, can take days, weeks, or months to run. Knowing how to predict the run time for a given program and set of inputs is critical to making research deadlines.

30.2 Analysis of Algorithms

Analysis of algorithms is one of the core areas of study in computer science. It is usually one of the main topics in multiple undergraduate courses, and in graduate courses.

Analysis of algorithms is an attempt to understand the computational costs of various algorithms in a mathematical sense. Algorithms are dissected and studied carefully to determine the relationship between inputs, the algorithm, and the number of computations required.

30.2.1 Order of Operations: Big-O

One of the basic tools in algorithm analysis is the Big-O notation, or order of operations.

Since computing algorithms can be extremely complex, and their exact implementation will vary across different languages and hardware, predicting the exact behavior of an algorithm is too difficult a task to pursue. Part of the analysis process is eliminating factors that will not contribute significantly to the run time.

For example, the run-time of a fairly simple algorithm may turn out to be $4n^2 - 5n + 4$ seconds, where $n$ is the number of elements in the input.

If $n$ is 100, then $4n^2$ is 40,000, -5n is -500, and 4 is just 4. We see here that the -5n and 4 account for only about 1% of the run time. As $n$ gets larger, they become even less significant, since $n^2$ grows much faster than $n$ as $n$ increases. Furthermore, we are most interested in run times when $n$ is large, and the run times are long. When $n$ is very small, the -5n and 4 terms may be more significant, but who cares? The program will complete in practically no time anyway.

In short, the dominant term in the run time formula when $n$ is large is $4n^2$. We therefore discard the other terms to simplify the process of predicting run time. We also discard the constant factor 4, since it will vary with the language and hardware, and therefore tells us nothing about the algorithm.

Finally, the actually run-time will depend on many factors that are difficult or impossible to predict, such as other load on the computer at the time it runs.

The algorithm itself determines what the run time will be proportional to, and the constant of proportionality will be determined empirically for each computer we run it on.

Hence, for this algorithm, we say the complexity is $O(n^2)$ ("order n squared"). This means that the run time is approximately proportional to $n^2$. 

---


30.2.2 Predicting Run Time

If we know the order of operations for an algorithm and have a single sample run time, we can predict run times for other input sizes fairly accurately, if certain assumptions hold.

If an $O(N^2)$ program takes 10 seconds to process 10,000 elements on a given computer, how long will it take to process 50,000?

1. time = $K \times n^2$
2. 10 seconds = $K \times 10,000^2$
3. $K = 10 / 100,000,000 = 10^{-7}$
4. time$_{50,000} = K \times 50,000^2 = 10^{-7} \times 50,000^2 = 250$ seconds

The assumption is that the value $K$ is the same for both values of $n$. This will only be true if the CPU and memory performance are the same. One case where this might not hold is when $n$ is sufficiently large to cause the system to run out of RAM and begin using swap space. On this case, memory performance will drop, and $K$ will increase.

30.2.3 Determining Order

The order of operations is relatively easy to determine for many algorithms, whereas the exact run time is nearly impossible. Determining the order of operations for a given algorithm requires carefully analyzing what the algorithm does.

Example 30.1 Selection Sort
Consider the selection sort algorithm. Sorting algorithms consist mainly of comparisons and swaps.
To find the smallest element in a list of $N$ elements, the selection sort must perform $N-1$ comparisons. It then swaps this element with the first element.
To find the next smallest element, it must perform $N-2$ comparisons, and then does another swap.
Continuing on, it will do $N-3$ comparisons and another swap, $N-4$ and another swap, until the last round where it does 1 comparison and 1 swap.
In all, it does $(N-1) + (N-2) + \ldots + 1$ comparisons, which is approximately $N^2/2$, and $N-1$ swaps. Since number of comparisons has the highest order, the algorithm is $O(N^2)$.

Example 30.2 Binary Search
Now consider searching a sorted list, such as a phone book. To conduct this search systematically, we might start in the middle, and see how the middle entry compares to the search key (the item we’re searching for). If the item matches the key, we’re done. If it is either less than or greater than the key, we have eliminated half the list as potential candidates. We then go to the middle of the half of the list that remains. We repeat this process until we either find the key, or we’re down to one element. This algorithm is known as the Binary Search.

So what is the order of operations? If the search key is in the list, the actual number of comparisons is a matter of luck. We could hit it on the first try, or any try after. This is only predictable by examining the entire list. What we are interested in, though is the worst and average case. The worst case occurs when the key is either not found, or is the last item we check. The number of comparisons is the number of times we can divide the list in half, which is $\log_2(n)$. ( $2^n$ = the size of a list if we start with one element and double the list size $n$ times.)

Hence, the order of operations is $O(\log(n))$. We drop the base 2, since it is a constant factor: $\log_2(n) = \ln(n) / \ln(2)$.

30.2.4 Common Algorithms

Most commonly used algorithms have been carefully studied and their orders of operation are well known.

Note that for multidimensional data, $n$ is taken to be a single dimension. For example, for a 100 x 100 matrix, $n$ is 100. For a non-square matrix, we can use $n = \sqrt{\text{rows} \times \text{cols}}$. 
### Algorithm

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Order of Operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Selection Sort</td>
<td>$O(n^2)$</td>
</tr>
<tr>
<td>Bubble Sort</td>
<td>$O(n^2)$</td>
</tr>
<tr>
<td>Quicksort</td>
<td>$O(n \log(n))$</td>
</tr>
<tr>
<td>Heapsort</td>
<td>$O(n \log(n))$</td>
</tr>
<tr>
<td>Matrix Addition</td>
<td>$O(n^2)$</td>
</tr>
<tr>
<td>Matrix Multiplication</td>
<td>$O(n^3)$</td>
</tr>
<tr>
<td>Traveling Salesman</td>
<td>$O(2^n)$</td>
</tr>
</tbody>
</table>

**Table 30.1: Common Orders of Operation**

Note that the Traveling Salesman algorithm has exponential order. The goal of the traveling salesman problem is to determine the shortest route for a salesman to visit all of $n$ cities once. It has been shown that to find the shortest path deterministically (with certainty) requires constructing every possible route. For $n$ cities, there are $2^n$ possible routes that visit each city once.

Any algorithm with an exponential order is considered *intractable* (unsolvable), even on a very fast computer, because the number of calculations required grows too quickly to make it possible to solve for large values of $n$. Even worse than exponential time are $O(n!)$ and $O(n^n)$.

When faced with problems like this, computer scientists may look for *heuristic* (probabilistic) solutions. A good heuristic solution won’t guarantee an optimal result, but will provide a near-optimal result with high probability.

### 30.3 Performance Measurement

The next question is how to obtain the sample run time needed to predict longer run times.

#### 30.3.1 Timing Programs

In Unix, the `time` command can be used to measure the actual time, CPU time, and system time of any program. To use it, we simply place the command "time" before the command to be benchmarked:

```
408: f90 -O selsort.f90 -o selsort
409: time selsort 50000nums > output
4.985u 0.044s 0:05.03 99.8% 10+1147k 0+12io 0pf+0w
```

This shows that the program used 4.98 seconds of user time (the program itself running), 0.044 seconds of system time (time spent by the operating system providing service to the program, such as reading the input file), and took 5.03 seconds of "wall clock time", or "real time" to complete from the moment the command was entered.

If the program is the only intensive process running, then real time should be about equal to user time + system time. If there are other programs running that compete for the CPU, memory, or disk, the real time could be significantly longer than the user + system time.

#### 30.3.2 Profiling: Timing Subprograms

The `time` command is an easy way to determine run time for an entire process, but what if we want to determine run time for an individual subprogram? Large programs may implement many algorithms, and the `time` command does not allow us to see how much time each of them is using.

Profiling is a feature of most Unix compilers that allows us to see how much time each subprogram used. To use profiling, we must first compile the program with the `-p` flag (or `-pg` for GNU compilers). This causes the compiler to insert code that checks the time before and after every subprogram call. The results of these measurements are saved in a file called `mon.out`. We then use the `prof` command to see the results.

In the profile below, we can see that the `sort_list()` subroutine used most of the CPU time in our selection sort program.
weise bacon ~ 413: f90 -O -p selsort.f90
weise bacon ~ 414: ./selsort 50000nums > output
weise bacon ~ 415: prof ./selsort

%Time Seconds Cumsecs #Calls msec/call Name
94.9 2.60 2.60 1 2600. sort_list_
2.2 0.06 2.66 __f90_slw_ia32
1.1 0.03 2.69 50000 0.0006 __f_cvt_real
0.7 0.02 2.71 __mt_prof_release_lock
0.4 0.01 2.72 __f90_open_for_output_r
0.4 0.01 2.73 50001 0.0002 __f90_sslr
0.4 0.01 2.74 50001 0.0002 __f90_eslw
0.0 0.00 2.74 1 0. main
0.0 0.00 2.74 1 0. read_list_
0.0 0.00 2.74 1 0. MAIN_
0.0 0.00 2.74 1 0. print_list_
0.0 0.00 2.74 3 0. __getarg_
0.0 0.00 2.74 1 0. __f90_deallocate
0.0 0.00 2.74 1 0. __f90_allocate2
0.0 0.00 2.74 1 0. f90_init
0.0 0.00 2.74 50001 0.0000 __f90_ssslw
0.0 0.00 2.74 1 0. __f90_slw_ch
0.0 0.00 2.74 50000 0.0000 __f90_slw_r8
0.0 0.00 2.74 1 0. __f90_init
0.0 0.00 2.74 1 0. __f90_slr_i4
0.0 0.00 2.74 50000 0.0000 __f90_slr_r8
0.0 0.00 2.74 50001 0.0000 __f90_eslr
0.0 0.00 2.74 50001 0.0000 __f90_get_default_output_unit
0.0 0.00 2.74 50000 0.0000 __f90_get_numbered_unit_a
0.0 0.00 2.74 1 0. __f90_release_unit
0.0 0.00 2.74 1 0. __f90_open
0.0 0.00 2.74 1 0. __f90_close_unit_a
0.0 0.00 2.74 1 0. __f90_close_a
0.0 0.00 2.74 1 0. __f90_close
0.0 0.00 2.74 1 0. __f90_initio_a
0.0 0.00 2.74 1 0. __f90_opencat

30.4 Homework

1. What does Big-O notation represent? How does it relate to program performance?

2. Most of the run-time for a given program results from an O(n^3) algorithm, where n represents the number of inputs. The program takes 10 seconds to process 1000 inputs. About how long will it take to process 5000 inputs?

3. What is the order of operations for a linear search?

4. What is the order of operations for a binary search?

5. What is the order of operations for a matrix addition (n x n matrix).
Chapter 31

Structures

Assigned readings: Sections 10.1 - 10.3

31.1 Motivation

31.2 C Structures

31.3 Fortran Structures

31.4 Classes and Object-oriented Programming (OOP)

31.5 Homework
Chapter 32

The Preprocessor

Assigned readings: This material is not in the textbook.
Use .F90 instead of .f90.

32.1 Defined Constants

32.2 Conditional Compilation

32.3 Homework
Chapter 33

Software Project Management

33.1 Building with Make

The make utility is a standard Unix program that automatically rebuilds a designated set of files when some other files have changed. The relationships between the files are spelled out in a makefile, which is usually called Makefile. The makefile indicates which files are built from other files, and contains the commands for performing the builds.

A makefile consists of a set of build rules in the following form:

```
target-file: source-file1 source-file2 ...
    command1
    command2
    ...
```

Each rule is interpreted as "if any source file is newer than target-file, then execute the commands". This is made possible by the fact that Unix records the last modification time of every file on the system.

⚠️ **Caution** One of the quirky things about make is that every command must be preceded by a tab character. There must be a tab character, and you cannot substitute spaces.

Make can be used to generate any files from any other files, but is most commonly used to build an executable from a group of source files. After editing a source file, the user runs:

```
mypc: make
```

By default, make looks for a file called Makefile and if present, executes the rules in it. A makefile with a different name can be specified following -f.

```
mypc: make -f myprog.mk
```

33.1.1 Building a Program

Suppose we want to build an executable program from two source files called myprog.f90 and matrix.c, using the GNU C and Fortran compilers.

To accomplish this, we must first compile each source file to an object file with the -c flag. When compiling with -c, a file is compiled (translated to an object file containing machine code), but not linked with other object files or libraries to create a complete executable. The object files have a ".o" extension.
After building all the object files, they are linked together along with additional object files from libraries to produce the complete executable. The simple Makefile below demonstrates:

```
# Link myprog.o, matrix.o and standard library functions to
# produce the executable myprog. gfortran knows how to link Fortran
# object files with C object files, while gcc would require additional
# flags.
myprog: myprog.o matrix.o
    gfortran -o myprog myprog.o matrix.o

# Compile myprog.f90 to an object file called myprog.o
myprog.o: myprog.f90 Makefile
    gfortran -c myprog.f90

# Compile matrix.c to an object file called matrix.o
matrix.o: matrix.c
    gcc -c matrix.c
```

If no target is specified on the command-line, make begins by trying to build the first target in the makefile. Hence, running `make` with no arguments will begin with the rule to link `myprog` from the object files. If one or more targets are given on the command-line, make tries to build only those targets (including any sources they depend on).

```
mpc: make matrix.o
```

Before building any target, `make` first checks to see if any of the sources are also targets in another rule. If so, that rule is executed first to ensure that the source is up-to-date. For example, before executing the rule to build `myprog`, `make` will search the makefile for rules to build `myprog.o` and `matrix.o`, and execute them first.

### 33.1.2 Make Variables

We can improve this makefile by using variables to eliminate redundant hard-coded commands and filenames:

```
BIN= myprog
OBJS= myprog.o matrix.o
CC= gcc
FC= gfortran
LD= ${FC}

# Link myprog.o, matrix.o and standard library functions to
# produce the executable myprog. gfortran knows how to link Fortran
# object files with C object files, while gcc would require additional
# flags.
$(BIN): $(OBJS)
    $(FC) -o $(BIN) $(OBJS)

# Compile myprog.f90 to an object file called myprog.o
myprog.o: myprog.f90 Makefile
    $(FC) -c myprog.f90

# Compile matrix.c to an object file called matrix.o
matrix.o: matrix.c
    $(CC) -c matrix.c
```

It can be tricky to write portable makefiles, since users may use different operating systems and different compilers. Some people solve this by using configuration scripts to generate a makefile from a more generic set of rules.

However, configure scripts are largely unnecessary if the makefile uses variables rather than hard-coded commands and options. Some might argue that large, complex software applications cannot be effectively represented by a simple makefile. However,
to borrow language from Alan Greenspan, a project that’s too big for a simple makefile is too big, and should be broken into smaller, independent projects.

Modern make utilities have a conditional assignment operator that only assigns the value if it was not provided on the command line or in the environment.

```bash
# Use gcc if not specified otherwise on the make command-line
# or in the environment
CC?= gcc
```

If Makefile contains the conditional assignment above, then the make command below will use gcc, unless CC is defined as an environment variable:

```bash
mypc: make
```

Either of the make commands below will use icc:

```bash
mypc: make CC=icc
```

```bash
mypc: setenv CC icc
mypc: make
```

A complete makefile using conditional assignments:

```bash
BIN= myprog
OBJS= myprog.o matrix.o

CC?= gcc
FC?= gfortran
LD= ${FC} # Use fortran compiler to link
CFLAGS?= -Wall -O
FFLAGS?= -Wall -O

# Link myprog.o, matrix.o and standard library functions to # produce the executable myprog. gfortran knows how to link Fortran # object files with C object files, while gcc would require additional # flags.
${BIN}: ${OBJS}
  ${LD} -o ${BIN} ${OBJS}

# Compile myprog.f90 to an object file called myprog.o
myprog.o: myprog.f90 Makefile
  ${FC} -c ${FFLAGS} myprog.f90

# Compile matrix.c to an object file called matrix.o
matrix.o: matrix.c
  ${CC} -c ${CFLAGS} matrix.c
```

Note Some variables used in makefiles, such as CC, FC, LD, CFLAGS, FFLAGS, and LDFLAGS, are standardized. They have special meaning to make and to ports/packages systems. Hence, you should not use different variable names to indicate compilers, linkers, and compile/link flags.

### 33.1.3 Special Targets

Some additional common targets are included in most makefiles to install the binary and to clean up files generated by the makefile. These targets usually have no associated source, which means they are always executed if specified.
BIN= myprog
OBJJS= myprog.o matrix.o
CC?= gcc
FC?= gfortran
LD= $(FC) # Use fortran compiler to link
CFLAGS?= -Wall -O
FFLAGS?= -Wall -O

# Where to install files
PREFIX?= /usr/local

# Link myprog.o, matrix.o and standard library functions to
# produce the executable myprog. gfortran knows how to link Fortran
# object files with C object files, while gcc would require additional
# flags.
$(BIN): $(OBJJS)
  $(LD) -o $(BIN) $(OBJJS)

# Compile myprog.f90 to an object file called myprog.o
myprog.o: myprog.f90 Makefile
  $(FC) -c $(FFLAGS) myprog.f90

# Compile matrix.c to an object file called matrix.o
matrix.o: matrix.c
  $(CC) -c $(CFLAGS) matrix.c

clean:
  rm -f $(BIN) $(OBJJS)

install:
  install -c $(BIN) $(PREFIX)/bin

33.1.4 Building Libraries

An almost identical makefile can be used to build a library instead of an executable. We need only replace the link command
with a command that build a library archive. Also, none of the source files used to build a library should contain a main program.

LIB= libmatrix.a
OBJJS= read.o write.o invert.o
CC?= gcc
CFLAGS?= -O -Wall
AR?= ar
RANLIB?= ranlib

# Build a static library
$(LIB): $(OBJJS)
  $(AR) r $(LIB) $(OBJJS)
  $(RANLIB) $(LIB)

read.o: read.c matlib.h Makefile
  $(CC) -c $(CFLAGS) read.c

write.o: write.c matlib.h Makefile
  $(CC) -c $(CFLAGS) write.c

invert.o: invert.c matlib.h Makefile
  $(CC) -c $(CFLAGS) invert.c
This library can then be used by any program by linking with `-lmatrix`:

```bash
BIN= myprog
OBJS= myprog.o
LIBS= libmatrix.a

${BIN}: ${OBJS} ${LIBS}
  ${LD} -o ${BIN} ${OBJS} -lmatrix

# Compile myprog.f90 to an object file called myprog.o
myprog.o: myprog.f90 Makefile
  ${FC} -c ${FFLAGS} myprog.f90
```

### 33.1.5 Mixing Compilers

Mixing modules compiled with different vendor’s compilers is not advisable. GNU, Intel, and Sun compilers, for example, use slightly different formats for the machine code files they output. Different versions of the same compiler brand, e.g. gcc 4.2.1 and gcc 4.4.3, may also be incompatible.

While it may be possible to build a program using modules compiled with different compilers, the subtle differences between those modules can produce problems that are very difficult to debug.

Users are therefore advised to build all program modules with the same version of the same compiler. If pre-compiled libraries are being used (e.g. MPI, lapack, etc.) be sure to use the same compiler that the libraries were built with. A slightly newer compiler is usually OK as well, but even that introduces some doubt about the stability of the executable.

---

**Caution** Note that ‘gcc’ and ‘cc’ are not necessarily interchangeable, even if your environment is configured to use the correct compiler. Some gcc installations may not have ‘cc’ linked to ‘gcc’, so using ‘cc’ may result in using the gcc version that ships with the OS, rather than the one you intended. On a Solaris system, ‘cc’ would represent the Sun compiler, even if gcc is installed.

---

### 33.1.6 Mixing C, C++, and Fortran: A Brief Overview

It is possible to build programs using a mixture of C, C++, and Fortran code. There are some pitfalls, however. The details of mixing languages are beyond the scope of this manual, but a few examples are provided here to introduce the general idea.

#### Data formats

One major issue for scientific programmers is the structure of multidimensional arrays. C and C++ use row-major format (all elements of a given row are stored contiguously in memory), while Fortran uses column-major format (all elements of a given column are contiguous).

Hence, `matrix[row][col]` in C or C++ is represented as `matrix(col,row)` in Fortran.

This is just one of many differences in how data structures differ between C/C++, and Fortran. More information can be found on the web by searching for "mixing C and Fortran”.

#### Libraries

Fortran compilers typically *decorate or mangle* symbol names with the program by appending an underscore character. Hence, a Fortran function called `invert()` would be called as `invert_()` from within a C or C++ program.
Calling C functions from within a Fortran or C++ program doesn’t generally require such name mangling. Hence, it is often easier to use the Fortran or C++ compiler for the link phase when building a mixed-language program that calls C functions.

This could, however, result in “undefined symbol” errors, since a Fortran compiler will not search the standard C libraries by default. If calling standard C library functions from a Fortran program, you may need to add `-lc` to the link command, so that the Fortran linker will search the standard C libraries.

The example makefiles below show how to build programs with various mixtures of C, C++, and Fortran. If you are not familiar with makefiles, you may want to read Section 33.1 first.

### Link
```
# Link
program: main.o functions.o
  gfortran -o program main.o functions.o -lc

main.o: main.f90
  gfortran -c main.f90

functions.o: functions.c
  gcc -c functions.c
```

If calling C++ library functions from Fortran, add `-lstdc++`.

```
# Link
program: main.o functions.o
  gfortran -o program main.o functions.o -lstdc++

main.o: main.f90
  gfortran -c main.f90

functions.o: functions.cc
  g++ -c functions.cc
```

If calling Fortran library functions from C++ with the GNU compiler collection, add `-lgfortran`.

```
# Link
program: main.o functions.o
  g++ -o program main.o functions.o -lgfortran

main.o: main.cc
  g++ -c main.cc

functions.o: functions.f90
  gfortran -c functions.f90
```

If using MPI, force the linker to search the C++ MPI libraries using `-lmpi_cxx`.

```
# Link
program: main.o functions.o
  mpif90 -o program main.o functions.o -lstdc++ -lmpi_cxx

main.o: main.f90
  mpif90 -c main.f90

functions.o: functions.cc
  mpic++ -c functions.cc
```
33.2 Software Deployment

Software management is the installation and deinstallation of programs and libraries on your system. Doing it well is very complex. It requires keeping track of dependencies between packages, including specific version requirements. It involves preventing conflicts between various packages and different versions of the same package. With the huge number of open source applications and libraries available today, we need sophisticated tools to manage our software installations and maintain a clean system.

Chapter 4 and Section 7.20 discuss the basics of software management from the end-users’ perspective. In this chapter, we will approach software development and management from the programmers’ perspective.

33.3 Version Control Systems

Git, Subversion

33.4 Source Code Hosting

Github, PyPI

33.5 Leave Dependencies to Package Managers

Many developers are tempted to create their own automated system to build and install their software and other programs and libraries required by it. This strategy, known as bundling inevitably creates extra work for developers and most of the end-users.

There are many problems with bundling:

- You cannot begin to imagine what your build system will encounter on the end-users’ systems. They may be running different operating systems and many of their systems are badly managed. Your build system will fail much of the time, leading to many inquiries from frustrated users.

  Attempts to help all of these users will result in a build system that grows in complexity without limit and becomes a major drain on your time. Many developers get frustrated with this situation and end up limited support to one or a few platforms. This is very unfortunate for the community, since different developers use different platforms, and end-users may not be able to install their programs on the same machine as a result, unless they resort to using virtual machines or other systems adding unnecessary overhead.

- Libraries you bundle may conflict with installed versions during build and/or at run time.

- Bundling dependencies also makes it more difficult for people to create packages for your software, since package managers are designed to handle things in a modular fashion.

  You might argue that you need a specific version of a library that is different from what’s available in your operating system’s package manager. It may actually be easier to make your software work with the mainstream version of the library than to bundle another version. Even if you really need an alternative version, it’s generally less problematic to create a separate package that can coexist with the mainstream version than it is to bundle the library with your software.

  If you need to modify a function in a dependent library, then rather than bundle the entire library, you could simply include the modified function in your code until your patches have been incorporated "upstream". If the linker finds the function in your program, it simply won’t look for it in the installed library. Hence, your patched version will take precedence.

Bundling is, in effect, inventing and maintaining your own package manager. If you do this, you will soon realize how complex the task is.

If you instead aim at making it easy to include your software in existing package managers, you will be free from all these headaches and able to focus on developing your code. Ways to do this are described in Section 33.8.
33.6 Avoid Custom Build Systems

Package managers like Debian packages, FreeBSD ports, Gentoo Portage, MacPorts, pkgsrc, etc. are designed to automatically install software and track dependencies. They are used and maintained by many people around the world and thus are very sophisticated and reliable.

Package managers provide by far the easiest way to install, uninstall, and upgrade software.

Unfortunately, many software developers attempt to replicate the functionality of package managers with esoteric, custom build systems designed only for their software and the software it depends on.

Almost none of them work well, because developers don’t have the time or resources to test them in any environment other than their own. They usually make it more difficult for end users to install your software. Your custom build system cannot come close to replicating the capabilities of an established package manager.

As a software developer, you can help yourself and end-users immensely by simply making it package-friendly, i.e. making it easy to incorporate your software into existing package managers. Ways to do this are described in Section 33.8.

Doing so leverages the work of thousands of other people, saving you a lot of work, and saving end users a lot of problems.

If you let package managers do what they’re meant for, even Makefile generators like CMake and GNU autotools are largely unnecessary. A simple Makefile that respects the environment, utilizing standard make variables like CC, CXX, FC, LD, CFLAGS, CXXFLAGS, FFLAGS, and LDFLAGS, will allow your software to be easily built by most package managers.

Alan Greenspan famously stated that any bank that’s too big to fail (without impacting the economy) is too big. Similar language can be used to describe software build systems: A project that is too big for a simple Makefile is too big, and should probably be broken into several smaller projects that can be built and installed independently of each other.

Package managers all want basically the same thing: The ability to control the building and installation of your software. If you provide a simple Makefile that works well with one package manager, it will be easy to port your software to any other. Your software will become easy to install and widely available with very little on your part.

For example, if you develop on a Debian based system and focus on supporting installing your software via Debian packages, you will learn how to make it easy to create a Debian package for your software. In doing so, you will fortuitously make it easy for yourself or others to create a Cygwin package, a FreeBSD port, a MacPort, an RPM, etc.

33.7 Follow the Filesystem Hierarchy Standard

Most Unix-compatible systems conform to the Filesystem Hierarchy Standard, as described on Wikipedia, the Linux hier man page and the FreeBSD hier man page.

This standard ensures that files such as programs, headers, libraries, configuration files, and documentation are easy to find for both people and software. Installing files where the system naturally looks for them relieves users of the need to modify their environment with problematic alterations to environment variables such as PATH and LD_LIBRARY_PATH. Altering these variables generally has unintended consequences for other programs, creating headaches for users and those who support the software.

The hierarchy standard also eliminates the need for programs to use clever tricks to locate their own files, such as Perl’s findbin. Some people believe that ignoring the hierarchy standard and installing every program into its own directory

Note that the hierarchy is relocatable: Different systems implement the hierarchy for add-on software under different prefixes. For example, by default, the Debian package system installs directly under /usr, the FreeBSD Ports system uses /usr/local, and MacPorts uses /opt/local.

Your Makefile should allow these systems to control the installation location by using standard variable names such as DESTDIR and PREFIX, e.g.

```
install:
  $(INSTALL) myprog $(DESTDIR)$(PREFIX)/bin
  $(INSTALL) myheader $(DESTDIR)$(PREFIX)/include
```

This is discussed in detail in Section 33.8.
33.8 Package-friendly Software

33.8.1 Modularity is the Key
TBD

33.8.2 All You Need is a Simple Makefile
TBD

33.8.3 Use Conventional Tools
TBD

33.8.4 Archiving Standards
TBD

33.8.5 Respect the Environment

Look for dependencies ONLY where told to. Do not hard-code search paths for libraries, etc.

Use standard compiler variables such as provided as environment variables or make variables.
CC, CFLAGS, CXX, CXXFLAGS, FC, FFLAGS, CPP (this is the C preprocessor, not C++ compiler!), CPPFLAGS

33.8.6 Install the Same Files Everywhere

TBD

33.8.7 A Simple Example

Debian Package
TBD

FreeBSD Port
TBD

MacPort
TBD

Pkgsrc Package
TBD
Part IV

Parallel Programming
Chapter 34

Parallel Programming

34.1 Serial vs. Parallel

The first rule in parallel programming: Don’t, unless you really have to.

34.1.1 Optimize the Serial Code First

Anything you can do in serial, you can do in parallel with a lot more effort.

People often look to parallelize their code simply because it’s slow, without realizing that it could be made to run hundreds of times faster without being parallelized. Using better algorithms, reducing memory use, or switching from an interpreted language to a compiled language will usually be far easier than parallelizing the code. Parallel programming is difficult, and running programs on a cluster is inconvenient compared to running them on your PC.

Some people are motivated by ego or even a pragmatic desire to impress someone, rather than a desire to solve a real problem. Using a $1,000,000 cluster to make inefficient programs run faster or to make yourself look smart is an unethical abuse of resources. If an organization spends this much money on computing resources, it’s important to ensure that they are used productively. Programmers have a responsibility to ensure that their code is well optimized before running it on a cluster.

34.1.2 Parallel Computing vs Parallel Programming

As explained in Chapter 35, you don’t necessarily need to write a parallel program in order to utilize parallel computing resources. In many cases, you can run simply multiple instances of a serial program at the same time. This is known as “embarrassingly parallel” computing. This is not only far easier than writing a parallel program, it also achieves better speedup in most cases, since there are no communication bottlenecks between the many processes. This type of parallel computing also scales almost infinitely. While some parallel programs can’t effectively utilize more than a few processors, embarrassingly parallel computing jobs can usually utilize thousands and achieve nearly perfect speedup. (Running N processes at once will reduce the total computation time by a factor of N.)

Thank to parallelize your entire computing project, not your program. If an individual run takes only hours or days, and you have to do many runs, then embarrassingly parallel computing will serve your needs very well. Parallelizing a program is only worthwhile when you have a very long running program (usually weeks or more) that will only be run a few times.

34.1.3 Don’t Do Development on a Cluster or Grid

You don’t need parallel computing resources to develop and test parallel programs.

Parallel code can be developed and tested on a single machine, even with a single core. This is much easier and faster than developing in the more complex scheduled environment of a cluster or grid, and avoids wasting valuable resources on test runs that won’t produce useful output.
You can control the number of processes used by OpenMP, MPI, and other parallel computing systems, even using more than one process per core.

Of course, you won’t be able to measure speed-up until you run on multiple cores, but that doesn’t matter through most of the development cycle.

Develop small test cases to run on your development and testing system, which could be a server, a workstation, or even your laptop. This will be sufficient to test for correctness, which is the vast majority of the development effort.

### 34.1.4 Self-test

1. What should always be done to a program before parallelizing it? Describe at least two reasons.
2. Explain the difference between parallel computing and parallel programming. Which implies the other? Explain.
3. What does embarrassingly parallel mean?
4. Describe at least two advantages of embarrassingly parallel computing vs other parallel computing paradigms.

### 34.2 Scheduling Compilation

#### 34.2.1 Why Schedule Compilation?

A cluster consists of one or more head nodes and a number of compute nodes. Users log into a head node, and submit jobs which then run on one or more compute nodes. Because a head node serves a different purpose than the compute nodes within a cluster, it necessarily has a different set of software installed, and potentially a slightly different runtime environment.

Because of the configuration differences between the head node and the compute nodes, it is possible that software configured and compiled on the head node may not work properly on the compute nodes. It is therefore advisable to configure and compile software on a compute node, to ensure that it will run in the exact same environment in which it was configured and compiled.

Multiple users compiling large programs on the head node would also create a heavy load on the head node that would impact other users.

Since compute nodes are allocated by the scheduler for computational processes, it would not be safe to simply choose a compute node manually and compile our software on it. Doing so could interfere with a scheduled job. Therefore, compilations should be scheduled like any other process that requires a compute node.

On a heterogeneous cluster or grid (different nodes have different CPU architectures or different operating systems) you may want to compile the program on every node as part of the execution job. If a program is going to run for hours or days, adding seconds or minutes of compilation time to each process is a trivial cost.

#### 34.2.2 Batch Serial and Batch Interactive

For large compilations, you may wish to schedule a batch serial job as outlined in Section 11.3.3. This method will create an output file containing screen output from the build commands. An example build script for a software package with a Makefile is shown below. Note that each package may have its own build method, so the commands in the script below will need to be changed for each package you build.

```bash
#!/bin/sh

# SLURM submit script for compilation

make
```

For smaller builds, you may prefer to watch the compiler output as it occurs, in which case a batch interactive job would be more suitable. Batch interactive jobs are described in Section 11.3.3.
#!/bin/sh -e
# -e terminates script if any command fails
 ./configure --prefix $HOME/myprogs
 make
 make install

Scheduling compilations this way will not introduce any measurable delays in compiling your code. Dispatching of batch-serial and interactive jobs generally occurs almost immediately, since there is almost always one free core available in the cluster.

### 34.2.3 Self-test

1. Why should compilation be done under the scheduler on a cluster or grid?

2. Write a script for the scheduler or your choice that submits a job for compiling the program `matrix.c` to an executable called `matrix`.

### 34.3 Further Reading

More information on parallel programming resources such as books, courses, and websites is available on cluster services website: [http://www4.uwm.edu/hpc/](http://www4.uwm.edu/hpc/)
Chapter 35

Programming for HTC

35.1 Introduction

High Throughput Computing (HTC) is the simplest and generally most economical form of distributed parallel computing. In high throughput computing, processes are dispatched to run independently on multiple computers at the same time. The processes are typically serial programs, not parallel programs, so HTC can be thought of as parallel computing without parallel programming.

HTC is sometimes referred to as embarrassingly parallel computing, since it is so easy to implement.

Usually, the same program runs on all computers with different data or inputs, but the definition of HTC is not limited to this scenario. If you are running multiple independent processes simultaneously on different computers, you’re using HTC.

HTC has several advantages:

• It’s the easiest type of parallelism to implement. No parallel programming is necessary. It’s just a matter of running a serial program in multiple places at the same time.

• It does not require a high-speed network or any other special hardware. HTC often utilizes lab computers across a college campus, or even home computers around the world.

• It scales almost infinitely. Since the processes are independent of each other, there is no communication overhead between them, and you can run as many processes as you have cores to run them on with nearly perfect speedup. That is, if you run 1000 processes at once, the computations will finish almost 1000 times faster than if you ran them one at a time.

35.1.1 Self-test

1. Define high throughput computing.

2. What is meant by embarrassingly parallel?

3. Describe three advantages of HTC over other types of parallel computing.
35.2 Parallelize as a Last Resort

While there will always be a need for parallel computing, the availability of parallel computing resources may tempt people to use them as a substitute for writing good code.

There is virtually no optimal software in existence. Most software in existence at any given moment can be made to run faster, and a significant percentage of it can be made to run orders of magnitude faster. Most performance issues can and should therefore be resolved by optimizing the software first.

Improving software is a more intelligent way to resolve performance issues wherever it’s possible. It will allow effective use of the software on a much wider variety of hardware, possibly including ordinary desktop and laptop machines. This is a much better situation than needing a cluster or grid to get your work done.

It is also the more ethical way to resolve issues where users are running on shared resources. Using tens of thousands of dollars worth of computer equipment to make inefficient software run in a reasonable amount of time is wasteful and foolish, and may delay the work of others who need those resources for more intelligent uses.

Before you consider the use of parallel computing, make sure you’ve done all you can to optimize your software, by choosing efficient algorithms, using compiled languages for the time-consuming parts of your code, and eliminating wasteful code.

Software performance is discussed in greater detail in Part III.

35.3 Common Uses for HTC

35.3.1 Monte Carlo Experiments

Monte Carlo experiments or simulations are so named because they use random inputs to a system to uncover the behavior of that system. (Monte Carlo is a famous gambling site in the French Riviera).

One of the Monte Carlo experiments that is easiest to understand yet quite paradigmatic is the calculation of \( \pi \). It begins with the simple calculation. If one has a circle with radius \( r \), the area is \( \pi r^2 \). A square that encompasses that circle has side length of \( 2r \) and area of \( 4r^2 \). Therefore the ratio of the area of the circle over the area of the square is \( \frac{\pi r^2}{4r^2} = \pi/4 \).

Monte Carlo calculation of \( \pi \) is the equivalent of throwing a dart at the square many times, then dividing the number of times it hit the circle by the total number of throws. If our dart thrower is bad enough to produce a spread of darts that is randomly and uniformly distributed across the entire square, that ratio would be \( \pi/4 \). A more skilled dart thrower would produce a non-uniform distribution, which would render the simulation useless.

This task can easily be broken down in an embarrassingly parallel fashion. Each calculation of the "dart throw" is independent of the others, so we can distribute the work to as many independent processes as we like.

The main trick with this and other Monte Carlo simulations is making sure that the input to the calculation is as random as it can be and of the correct statistical distribution (uniform in this case). While it is impossible to generate truly random numbers on a computer, generation of pseudo-random numbers is well developed and provided by most programming languages. Calculation of large groups of pseudo-random numbers that is as close to truly random as possible is a major part of Monte Carlo simulation.

---

**Caution** The standard C libraries include older random number functions `rand()` and `srand()`. These functions are considered obsolete because they produce relatively predictable and frequently repeating sequences. They have been superseded by `random()` and `srandom()`, which generate much higher quality pseudo-random sequences. Many other languages also offer multiple pseudo-random number generators of varying quality, so be sure to find out which ones will work best for your purposes.

---

This task is easily broken up into independent smaller tasks. Hence, a simple approach to improve our results without taking more time would be running many simulations on different computers at the same time, and then averaging the results of all of them.

However, since pseudo-random number generators follow a predictable sequence, we have to be careful with this approach.
If we split this job into N processes, each one generating 500,000 random points, and the pseudo-random number generator started with the same value for each, then each process would generate the exact same pseudo-random number sequence and ultimately the exact same estimate for pi! Averaging the results of this parallel simulation would therefore produce exactly the same answer as each individual process.

The solution to this problem is to use a different seed for each process. The seed serves as a starting point for the pseudo-random number sequence, so if we use a different seed for each process, we will get different results. In C, the pseudo-random numbers generated are integers, so some commonly used seeds are the current system clock and the process ID.

```
srandom((unsigned int)time(NULL));
srandom((unsigned int)getpid());
```

Below is a simple C program that approximates pi given a number of "dart throws".

```c
#include <stdio.h>
#include <math.h>
#include <stdlib.h>
#include <sysexits.h>
#include <time.h>
#define TOTAL_POINTS 100000000

int main(int argc, char *argv[]) { 
  int i;
  int inside_circle;

  double x;
  double y;
  double distance_squared;

  // Initialize counters
  inside_circle = 0;

  // Initialize pseudo-random number sequence with a different value
  // each time
  srandom(time(NULL));

  // Compute X random points in the quadrant, and compute how many
  // are inside the circle
  for (i = 0; i < TOTAL_POINTS; i++) {
    // random() and RAND_MAX are integers, so integer division will
    // occur unless we cast to a real type
    x = (double)random() / RAND_MAX;
    y = (double)random() / RAND_MAX;

    // No need to compute actual distance. We need only know if
    // it's < 1 and it will be if distance_squared < 1.
    distance_squared = x * x + y * y;
  }

  return EXIT_SUCCESS;
}
```
if (distance_squared < 1.0)
    inside_circle++;

printf("Inside circle: %d Total: %d PI ~= %f\n",
    inside_circle, TOTAL_POINTS, (double)inside_circle / TOTAL_POINTS * 4);

return EX_OK;
}

---

Caution
The time() function has a resolution of seconds, so assuming the clocks of all the computers are in sync, it is likely that it will return the same value to many of the processes in a job array, which are all started at about the same time. Hence, many processes would produce duplicate results, rendering all but one of them useless.

Although very unlikely, the process ID returned by getpid() could by sheer coincidence be the same for processes running on different computers, since it is the Unix PID, not scheduler job/task/process ID.

For job arrays, using the scheduler job/task/process ID is a simple way to ensure a different value for every process.

Shown below is another version of the C program, calcpi-parallel.c. There are three command-line arguments as inputs:

```
shell-prompt: ./calcpi-parallel points process-index total-processes
```

We made the number of points a command-line argument rather than a constant in the program so that we don’t have to recompile in order to run a different experiment.

The process index provides a different seed to each process. The total-processes argument is not necessary, but is useful in the program to check for sanity errors in the command such as in invalid process index. Requiring a little redundancy from the end-user can often help catch mistakes that would be hard to track down later.

The output of calcpi-parallel is the counts of points inside the circle and the total number of points. We need a separate program that averages all this output to approximate pi.

```
天上七人 ** Description:
* Estimate PI using Monte Carlo method. Multiple instances of
  this program are run in parallel, and results of all are used
  to estimate the value of PI.
*
* ** Arguments:
  * argv[1]: index for this process
  * argv[2]: total number of processes
*
* ** Returns:
  * NA
*
* ** History:
  * Date        Name         Modification
    2011-08-15  Lars Olson    Begin
    2011-09-27  Jason Bacon   Replace rand() / srand() with random() / srandom()
    2012-06-29  Jason Bacon   Use different seed for each process to simplify
                           code. Add comments and use more descriptive
                           variable names.
**************************************************************************/

#include <stdio.h>
#include <math.h>
#include <stdlib.h>
```
#include <sysexits.h>

void usage(char *progname)
{
    fprintf(stderr, "Usage: %s total-points process-index total-processes\n", progname);
    exit(EX_USAGE);
}

int main(int argc, char *argv[])
{
    int process_index;
    int total_processes;
    // Unsigned long can be either 32 or 64 bits and is limited to about
    // 4.3 billion if the compiler defaults to 32-bits.
    // Unsigned long long (64 or 128 bits) allows the number of points
    // to be at least 2^64-1, or 1.84*10^19.
    unsigned long long total_points,
        inside_circle,
        i;

double x;
    double y;
    double distance_squared;

    char *end_ptr;

    // Make sure program is invoked with 2 command line arguments
    // argc = 3 for program name + arguments
    if ( argc != 4 )
        usage(argv[0]);

    // First command line argument is which job out of the 10 it is
    total_points = strtouq(argv[1], &end_ptr, 10);
    // Make sure whole argument was a valid integer
    if ( *end_ptr != '\0' )
    {
        fprintf(stderr, "Error: Argument 1 (%s) is not an integer.\n", argv[1]);
        usage(argv[0]);
    }

    // First command line argument is which job out of the 10 it is
    process_index = strtol(argv[2], &end_ptr, 10);
    // Make sure whole argument was a valid integer
    if ( *end_ptr != '\0' )
    {
        fprintf(stderr, "Error: Argument 1 (%s) is not an integer.\n", argv[1]);
        usage(argv[0]);
    }

    // Total number of processes in the job, 10 in our examples.
    total_processes = strtol(argv[3], &end_ptr, 10);
    // Make sure whole argument was a valid integer
    if ( *end_ptr != '\0' )
    {
        fprintf(stderr, "Error: Argument 2 (%s) is not an integer.\n", argv[2]);
        usage(argv[0]);
    }
if ( total_processes < 1 )
{
    fputs("Total processes must be > 0.\n", stderr);
    exit(EX_USAGE);
}

if ( (process_index < 1) || (process_index > total_processes) )
{
    fputs("Process index must be between 1 and total processes.\n", stderr);
    exit(EX_USAGE);
}

// Use a different seed for each process so that they don't all
// compute the same pseudo-random sequence!
srandom(process_index);

// Initialize counters
inside_circle = 0;

// Compute X random points in the quadrant, and compute how many
// are inside the circle
for (i = 0; i < total_points; i++)
{
    // random() and RAND_MAX are integers, so integer division will
    // occur unless we cast to a real type
    x = (double)random() / RAND_MAX;
    y = (double)random() / RAND_MAX;

    // If distance squared < or > 1.0, then distance < or > 1.0, so don't
    // waste time calculating the square root.
    distance_squared = x * x + y * y;
    if (distance_squared < 1.0)
        inside_circle++;
}

// %q wants unsigned long long, which can be 64 or 128 bits
// Cast the uint64_t variables to silence gcc warnings
printf("%llu %llu\n", inside_circle, total_points);

// Debugging only: not normal output used by scripts
// Comment out before running job
// printf("%f\n", (double)inside_circle / total_points * 4.0);

return EX_OK;

Software Performance Optimization

Software efficiency is a huge part of high performance and high throughput computing.
Improvements to software often result in an order of magnitude or more reduction in run time and in many cases make the use of parallel computing unnecessary.
Optimizing software before consuming expensive parallel computing resources can therefore be an ethical matter in many cases.
A little understanding of computer hardware can go a long way toward improving performance.
One prime example is understanding the difference between integers and floating point. Floating point types such as float and
double (real and double precision in Fortran) are inherently more complex than integers and therefore require more time for operations such as addition and multiplication.

Floating point types are stored in a format similar to scientific notation. As you may recall, adding scientific notation values is a three-step process:

1. Equalize the exponents
2. Add the mantissas
3. Normalize the result

Hence, we might expect floating point addition to take about three times as long as integer addition. Due to optimizations in floating point hardware, the difference is not that great, but it can be significant.

It pays to examine your program code and consider whether the use of floating point is really necessary. In most cases, including our calcpi program, it is not. The random() function in C produces integer values between 0 and RAND_MAX, a constant defined in the header file stdlib.h. We can just as easily perform these calculations using only integers, if we use quadrant dimensions of RAND_MAX x RAND_MAX instead of 1 x 1 and reduce the units of X and Y, so that all of our X and Y values are integers between 0 and RAND_MAX.

A modified version of our calcpi program using only integers is shown below:

```c
#include <stdio.h>
#include <math.h>
#include <stdlib.h>
#include <sys/exits.h>
#include <inttypes.h>

void usage(char *progname)
{
    fprintf(stderr, "Usage: %s total-points process-index total-processes\n", progname);
    exit(EXIT_USAGE);
}

int main(int argc, char *argv[])
```

* Description:
  * Estimate PI using Monte Carlo method. Multiple instances of
  * this program are run in parallel, and results of all are used
  * to estimate the value of PI.

* Arguments:
  * argv[1]: index for this process
  * argv[2]: total number of processes

* Returns:
  * NA

* History:
  * Date   Name       Modification
  * 2011-08-15 Lars Olson Begin
  * 2011-09-27 Jason Bacon Replace rand()/srand() with random()/srandom()\n  * 2012-06-29 Jason Bacon Use different seed for each process to simplify
  *        code. Add comments and use more descriptive
  *        variable names.

***************************************************************************/
int process_index;
int total_processes;

// Unsigned long can be either 32 or 64 bits and is limited to about
// 4.3 billion if the compiler defaults to 32-bits.
// Unsigned long long (64 or 128 bits) allows the number of points
// to be at least 2^64-1, or 1.84*10^19.
// The printf() function has no standard placeholders for uint64_t,
// so we'll use unsigned long long here to avoid compiler warnings.
// This won't affect performance much, since these variables are only
// incremented in the main loop.
unsigned long long total_points,
inside_circle,
i;

// random() returns 32-bit values. The square of a 32-bit value can
// be up to 64-bits long, so make sure the program uses 64-bit integers
// on all processors and for all calculations.
uint64_t x;
uint64_t y;
uint64_t distance_squared,
// RAND_MAX is a 32-bit integer, so cast it to 64 bits
// before multiplying to avoid a truncated result.
rand_max_squared = (uint64_t)RAND_MAX * (uint64_t)RAND_MAX;

char *end_ptr;

// Make sure program is invoked with 2 command line arguments
// argc = 3 for program name + arguments
if ( argc != 4 )
    usage(argv[0]);

// First command line argument is which job out of the 10 it is
total_points = strtouq(argv[1], &end_ptr, 10);

// Make sure whole argument was a valid integer
if ( *end_ptr != '\'0' ' )
{
    fprintf(stderr, "Error: Argument 1 (%s) is not an integer.\n", argv[1]);
    usage(argv[0]);
}

// First command line argument is which job out of the 10 it is
process_index = strtol(argv[2], &end_ptr, 10);

// Make sure whole argument was a valid integer
if ( *end_ptr != '\'0' ' )
{
    fprintf(stderr, "Error: Argument 1 (%s) is not an integer.\n", argv[1]);
    usage(argv[0]);
}

// Total number of processes in the job, 10 in our examples.
total_processes = strtol(argv[3], &end_ptr, 10);

// Make sure whole argument was a valid integer
if ( *end_ptr != '\'0' ' )
{
    fprintf(stderr, "Error: Argument 2 (%s) is not an integer.\n", argv[2]);
    usage(argv[0]);
}
if ( total_processes < 1 )
{
    fputs("Total processes must be > 0.\n", stderr);
    exit(EX_USAGE);
}

if ( (process_index < 1) || (process_index > total_processes) )
{
    fputs("Process index must be between 1 and total processes.\n", stderr);
    exit(EX_USAGE);
}

// Use a different seed for each process so that they don’t all
// compute the same pseudo-random sequence!
srandom(process_index);

// Initialize counters
inside_circle = 0;

// Compute X random points in the quadrant, and compute how many
// are inside the circle
for (i = 0; i < total_points; i++)
{
    x = (uint64_t)random();
    y = (uint64_t)random();

    // If distance squared is < or > randmax_squared, then distance is
    // < or > randmax, so don’t waste time calculating the square root.
    distance_squared = x * x + y * y;
    if (distance_squared < rand_max_squared)
        inside_circle++;
}

// %q wants unsigned long long, which can be 64 or 128 bits
printf("%llu %llu\n", inside_circle, total_points);

// Debugging only: not normal output used by scripts
// Comment out before running job
// printf("%f\n", (double)inside_circle / total_points * 4.0);

return EX_OK;
}

Timing this program on an AMD64 system shows that it is more than twice as fast as the earlier floating point version.

Note, however, that this calcpi requires the use of 64-bit integers. 32-bit processors can only do integer operations of 32 bits at a
time, although most can to 64-bit floating point operations all at once.

When using 64-bit integers on a 32-bit processor, we don’t see the same benefit, since the 32-bit processor has to do the 64-bit
integer operations in two steps. However, on a 32-bit Pentium 4 processor, the integer calcpi still performed about 25% faster
than the floating point version. If you’re looking at a month of calculations, the switch to integer arithmetic would still knock off
about a week.

Furthermore, many programs can get by with 32-bit integers, in which case you’ll realize the full performance benefit even on
32-bit hardware.

Calcpi on a SLURM Cluster

When compiling a program on a cluster or grid, it should be compiled on the same node(s) it will run on to ensure that it is
built and run in the exact same environment and in the presence of the exact same libraries and compiler tools. The head node
necessarily has different software installed, which may cause a program to be compiled differently than it would be on a compute node. Hence, we use a simple sbatch script to do the compilation:

```bash
#!/bin/sh -e
# Show commands in output
set -x
cc -O -o calcpi-parallel-integer calcpi-parallel-integer.c
```

To run the program in embarrassingly parallel fashion, we use a facility in SLURM and other queuing systems to run multiple instances of a single program, where each can receive different command line arguments and/or inputs. Consider the following SLURM submit script:

```bash
#!/bin/sh -e
#SBATCH --array=1-20
#SBATCH --output=calcpi-parallel-%a.out
#SBATCH --error=calcpi-parallel-%a.err
points=1000000000
./calcpi-parallel-integer $points $SLURM_ARRAY_TASK_ID 20
```

This script starts up an embarrassingly parallel job, known as a job array. The submit script is executed almost simultaneously on multiple cores in the cluster, one for each value in the range specified with --array.

The element $$\$$SLURM\_ARRAY\_TASK\_ID$$\$$ is a reference to an environment variable which is set by the SLURM scheduler to a different value for each job in the job array. With --array=1-10, SLURM\_ARRAY\_TASK\_ID will be 1 for one process, 2 for another, and so on up to 10.

Each process in the job array will also produce a separate output and error file with the value of SLURM\_ARRAY\_TASK\_ID appended. I.e., there will be files called calcpi-parallel-1.out, calcpi-parallel-2.out, etc.

After all the processes in the job array complete, the output can be tallied using a simple script such as the following:

```bash
#!/bin/sh
# If this is non-trivial, it should be done by submitting another job
# rather than run on the submit node.
# Results are on last line of each output file calcpi-parallel.out-1, etc.
# Both the SLURM and HTCondor scripts must name their output files to match this script.
# Send last line of each output file through a simple awk script that
# sums up each column and prints the quotient of the two sums.
tail -1 -q calcpi-parallel-*.out | awk \\
'BEGIN {
    in_circle=0;
    total=0;
}
{
    in_circle+=\$1;
    total+=\$2;
}
END {
    printf("%.2f\n", in_circle / total * 4.0);
}'
```
Another script (to be run directly, not submitted to the scheduler) could be used to clean up from previous runs and submit the job again:

```
#!/bin/sh -e

##########################################################################
# Script description:
#   Run a set of SLURM jobs to estimate PI, and combine the results.
#   An alternative to this approach is a DAG.
# Arguments:
#   None.
# Returns:
#   0, or status of first failed command.
# History:
# Date   Name    Modification
# 2013-01-29 Jason Bacon  Begin
##########################################################################

##########################################################################
# Main
##########################################################################

# Remove old output and log files
./clean

# Build program
sbatch calcpi-parallel-build.sbatch

# Submit computational jobs
sbatch calcpi-parallel-run.sbatch

Finally, the directory can be cleaned up using yet another script that we run directly:

```
#!/bin/sh

rm -f calcpi-parallel *.out* *.err* calcpi-parallel.[0-9]* calcpi-parallel-condor.log
```

Calcpi on an HTCondor Grid

Recall from Chapter 13 that HTCondor is a scheduling tool like SLURM, PBS or LSF. Unlike most schedulers, which are geared toward HPC clusters built entirely with dedicated hardware, HTCondor is specifically designed for grids that utilize a variety of hardware owned by a variety of people or departments, such desktop machines throughout your institution.
On the cluster, where all compute nodes run the same operating system and have access to the same files, we ran a separate build script to compile our code on one node, and then ran that executable on all nodes.

Since grids do not typically have shared file space, and are often heterogeneous (use different hardware and/or operating systems on various execute hosts), our approach to running code must be different.

If we are running software that’s preinstalled on the HTCondor compute hosts, as is often the case with programs like Octave and R, we must be aware that it may not behave identically on all hosts. Some hosts may be running a 32-bit operating system, limited to a few gigabytes of RAM, while others run 64-bit operating systems. There may be different versions of the software installed on different hosts.

These are additional issues we may need to deal with when using an HTCondor grid, but they are typically not difficult to resolve.

**Note** In HTCondor lingo, an *execute host* is equivalent to what we call compute nodes in a cluster.

Since we typically have no shared file system, we may need to transfer the program and input files to every execute host we use, and bring back any output files when the processes are done. All of this is handled automatically by HTCondor.

Since the execute hosts may have different hardware and operating systems, we can’t compile the program on one of them and expect it to run on all of them. Instead, it’s a common practice to transfer the source code to each execute host and compile it there as part of the job. This ensures that every binary file is compatible with the host it’s running on.

An HTCondor description file for running `calcpi-parallel.c` might appear as follows:

```plaintext
############################################################################
# Sample condor submit description file.
#
# Use \ to continue an entry on the next line.
#
# You can query your jobs by command:
# condor_q
############################################################################
# Choose which universe you want your program is running with
# Available options are
#  - standard:
#     Defaults to transfer executables and files.
#     Use when you are running your own script or program.
#
#  - vanilla:
#  - grid:
#     Explicitly enable file transfer mechanisms with
#     ‘transfer_executable’, etc.
#     Use when you are using your own files and some installed on the
#     execute hosts.
#
#  - java:
#     Explicitly enable file transfer mechanism. Used for java jobs.
#
#  - scheduler
#  - local
#  - parallel:
#     Explicitly enable file transfer mechanism. Used for MPI jobs.
#  - vm
# Refer http://research.cs.wisc.edu/condor/manual/v7.6/2_4Road_map
# Running.html for details.

universe = vanilla
```
# Macros (variables) to use in this submit description file
Points = 1000000000
Process_count = 100

# Specify the executable filename. This can be a binary file or a script.
# NOTE: The POVB execute hosts currently support 32-bit executables only.
executable = calcpi-parallel-condor.sh
# Command-line arguments for the execute command
# arguments =
# Set environment variables for use by the executable on the execute hosts.
# Enclose the entire environment string in quotes.
# A variable assignment is var=value (no space around =).
# Separate variable assignments with whitespace.
environment = "Process=$(Process) Process_count=$(Process_count) Points=$(Points)"

# Where the standard output and standard error from executables go.
# $(Process) is current job ID.
# We run calcpi-parallel under both SLURM and Condor, so we use same output
# names here as in calcpi-parallel-run.sbatch so that we can use the same
# script to tally all the outputs from any run.
output = calcpi-parallel-$(Process).out
error = calcpi-parallel-$(Process).err

# Logs for the job, produced by condor. This contains output from
# Condor, not from the executable.
log = calcpi-parallel-condor.log

# Custom job requirements
# Condor assumes job requirements from the host submitting job.
# IT DOES NOT DEFAULT TO ACCEPTING ANY ARCH OR OPSYS!!!
# For example, if the jobs is submitted from peregrine, target.arch is
# "X86_64" and target.opsys is "FREEBSD8", which do not match
# POVB execute hosts.
# You can query if your submitting host is accepted by command:
# Condor_q -analyze

# Memory requirements in megabytes
request_memory = 1000

# Requirements for a binary compiled on 32-bit CentOS 4 (POVB hosts):
# requirements = (target.arch == "INTEL") && (target.opsys == "LINUX")

# Requirements for a Unix shell script or Unix program compiled on the
# execute host:
requirements = ((target.arch == "INTEL") || (target.arch == "X86_64")) && 
             ((target.opsys == "FREEBSD") || (target.opsys == "LINUX"))

# Requirements for a job utilizing software installed via FreeBSD ports:
# requirements = ((target.arch == "INTEL") || (target.arch == "X86_64")) && \
# (target.opsys == "FREEBSD")

# Explicitly enable executable transfer mechanism for vanilla universe.
# true | false
transfer_executable = true

# yes | no | if_needed
should_transfer_files = if_needed

# All files to be transferred to the execute hosts in addition to the # executable.
transfer_input_files = calcpi-parallel.c

# All files to be transferred back from the execute hosts in addition to # those listed in "output" and "error".
# transfer_output_files = file1,file2,...

# on_exit | on_exit_or_evict
when_to_transfer_output = on_exit

# Specify how many jobs you would like to submit to the queue.
queue $(Process_count)

The executable script:

#!/bin/sh -e

# Use Bourne shell (/bin/sh) to maximize portability of this script. # There is no guarantee that other shells will be installed on a given host.

# Use /bin/sh -e to quit on the first error, so we don’t try to run # subsequent commands after a command in this script already failed.

# This script is meant to run on any Unix system (FreeBSD, Linux, Solaris, # etc.), so use only generic, portable Unix commands and flags. # (e.g. cc instead of gcc or icc).

# Bourne shell has a bare-bones PATH that won’t find some add-on software. # Also, startup scripts are not executed by Bourne shells under condor, # so local additions to PATH in /etc/* and ~/. will not be picked up. # If you know the path of programs you use on certain hosts, add it here.
# /usr/libexec Dependencies for some standard tools
# /usr/local/bin FreeBSD ports
# /opt/local/bin MacPorts
# /sw/bin Fink
PATH=$PATH:/usr/libexec:/usr/local/bin:/opt/local/bin:/sw/bin
export PATH

# Some slots may be on the same execute node so use a different executable # name for each process. Compilation could fail if both compiles try # to write the same executable file at the same time.
cc -o calcpi-parallel.$Process calcpi-parallel.c

# Condor process IDs start at 0, and calcpi-parallel expects indexes # to start at one.
./calcpi-parallel.$Process $Points $((Process+1)) $Process_count
If we have designed our scripts carefully, so that they use the same output filename, etc., then we can use the same tally and cleanup scripts that we used with SLURM:

```bash
#!/bin/sh

# If this is non-trivial, it should be done by submitting another job
# rather than run on the submit node.

# Results are on last line of each output file calcpi-parallel.out-1, etc.
# Both the SLURM and HTCondor scripts must name their output files to match
# this script.

# Send last line of each output file through a simple awk script that
# sums up each column and prints the quotient of the two sums.

tail -1 -q calcpi-parallel-*.out | awk \
    'BEGIN {
        in_circle=0;
        total=0;
    }
    {
        in_circle+=$1;
        total+=$2;
    }
    END {
        printf("%0.20f\n", in_circle / total * 4.0);
    }'

#!/bin/sh

rm -f calcpi-parallel *.out* *.err* \n    calcpi-parallel.[0-9]* \n    calcpi-parallel-condor.log
```

A simple shell script can also be used to automate the execution of the HTCondor jobs:

```bash
#!/bin/sh

# This script should be used to submit the HTCondor jobs.

# Script description:
# Run a set of condor jobs to estimate PI, and combine the results.
# An alternative to this approach is a DAG.

# Arguments:
# None.

# Returns:
# 0, or status of first failed command.

# History:
# Date       Name        Modification
# 2013-01-29 Jason Bacon Begin
```
# Remove old output and log files
./clean

# Submit computational jobs
condor_submit calcpi-parallel.condor

As an alternative to the shell script above, there is also a companion tool for HTCondor called DAGMan, which can be used to automate work flows (sequences of HTCondor jobs and other tasks). DAGMan uses DAGs (Directed Acyclic Graphs) to represent work flows. Some users may find this visual approach preferable to a script, especially for complex work flows.

### 35.3.2 Parameter Sweeps and File Transfer

One way in which the previous calcpi job is efficient is that it minimizes the amount of traffic passed throughout the cluster or grid. Ideally, only $N$ small files each with 2 integers in them were passed back to the head node or written to the shared file system of the cluster or grid. This was made possible by the fact that the calcpi program generates its own input in the form of pseudo-random numbers.

That kind of efficiency is not always possible and oftentimes one needs to pass input files from the head node to those working on the jobs.

An example of this is illustrated by a class of problems known as parameter sweeps. A parameter sweep is similar to a Monte Carlo simulation, in that the same program is run on a variety of inputs. However, the inputs in a parameter sweep are not usually random, nor are they necessarily even computable by the program. The inputs might be a range of values, and hence computable, or they may consist of vast amounts of previously collected or generated data stored in files.

In some cases, we know the correct output and need to find the input or inputs that will produce it. An example of this would be password cracking. Passwords are stored in a non-reversible encrypted form, which is often easy to obtain by reading a password file or eavesdropping on a network connection. It is not possible to decrypt a password directly (convert the encrypted form back to the raw form). However, it is relatively easy to encrypt (convert the raw form to the encrypted form), although such one-way encryption algorithms are deliberately designed to be expensive in order to slow down parameter sweeps aimed at password cracking.

In other cases, we might be given a vast amount of data collected by a scientific instrument such as a telescope, a DNA sequencer, or a microphone. Vast amounts of data could also come in the form of a large collection of electronic documents such as research papers, court documents, or medical records.

In cases like this, we will need to distribute the data to the compute nodes so that each process can work on a portion of the inputs.

When working on a cluster with a shared file system, we may be able to get away with leaving all the data in a shared directory where all of the processes can access it. If the amount of data and the number of processes working on it are very large, however, access to the shared file system can become a serious bottleneck. This will not only defeat the purpose of parallel computing for you, but will also seriously impact other users on the cluster.

Prestaging is especially important if the data files will be read more than once. However, even if they’ll only be read once, prestaging properly will prevent impacting the shared file system while the job runs.

Note, however, that individual disks on compute nodes are typically slower than the RAID arrays used for shared space. Also, compute nodes with many cores may have processes competing for the same local disks. All these factors must be considered when deciding how to prestage the data and how to distribute the processes across the nodes. For example, when running a data-intensive parallel job in a SLURM environment, we might ask the scheduler to run one process per node to ensure that each process is using a different local disk.

On a grid with no shared file system, prestaging the data is a requirement in all cases. For this reason, HTCondor has built-in facilities for file transfer both to and from the execute hosts.
Calcpi-file on a SLURM Cluster

For the sake of efficient learning, we will reuse the calcpi example to illustrate file transfer. In reality, no one would choose to provide inputs to calcpi this way, but using a familiar example should make it easier for the reader to follow, since new material is limited to just the process of using input files.

For this example we will calculate the random numbers first, and stores them in N files. Then we submit an array of N processes running a modified calcpi program that reads an input file instead of generating its own random numbers and outputs the same two numbers as the previous calcpi program does. There are several steps in this overall work flow:

1. Create N data files containing different pseudo-random number sequences.
2. Create a new submission script to run a job array with N different file names as inputs.
3. If there is no shared file system, or if using a shared file system would cause a bottleneck, prestage the N files to the compute nodes (or execute hosts).
4. Submit the job(s).
5. Tally the results.

This new work flow requires a new program called calcpi-file.c.

```c
/* ***************************************************************************
 * Description:
 * Estimate PI using Monte Carlo method.
 * *
 * Returns:
 * NA
 * *
 * History:
 * 2012-07-07  Jason Bacon  Derived from calcpi-parallel.c
 */

#include <stdio.h>
#include <math.h>
#include <stdlib.h>
#include <string.h>
#include <sys/exits.h>

void usage(char *progname)
{
    fprintf(stderr, "Usage: %s filename\n", progname);
    exit(EX_USAGE);
}

int main(int argc, char *argv[])
{
    unsigned long long i,
    inside_circle,
    total_points;
    extern int errno;

    double x;
    double y;
    double distance_squared;
    char *filename;
```
FILE *points_file;

if ( argc != 2 )
    usage(argv[0]);

// No need to copy the string, just point to it
filename = argv[1];

points_file = fopen(filename, "r");
if ( points_file == NULL )
{
    fprintf(stderr, "Could not open %s: %s\n", filename, strerror(errno));
    exit(EX_NOINPUT);
}

if ( fscanf(points_file, "%llu\n", &total_points) != 1 )
{
    fprintf(stderr, "Failed to read total_points from %s: %s\n", filename, strerror(errno));
    exit(EX_NOINPUT);
}

// Initialize counters
inside_circle = 0;

// Compute X random points in the quadrant, and compute how many // are inside the circle
for (i = 0; i < total_points; ++i)
{
    if ( fscanf(points_file, "%lg %lg\n", &x, &y) != 2 )
    {
        fprintf(stderr, "Failed to read point %llu from %s: %s\n", i, filename, strerror(errno));
        exit(EX_NOINPUT);
    }
    distance_squared = x * x + y * y;
    if (distance_squared < 1.0)
        inside_circle++;
}

// Error checking an fclose() is really only useful when writing to // a file, but we do it anyway to promote good habits.
if ( fclose(points_file) != 0 )
{
    fprintf(stderr, "Failed to close %s: %s\n", filename, strerror(errno));
    exit(EX_NOINPUT);
}

printf("%llu %llu\n", inside_circle, total_points);

return EX_OK;

This program is executed as follows:
peregrine: ./calcpi-file rand1.txt

An example input file:
Note This program doesn’t need to receive the number of data points as an argument. That information is determined by reading the input file.

The input files are generated by a separate program:

```c
#include <stdio.h>
#include <math.h>
#include <stdlib.h>
#include <sysexits.h>
#include <time.h>
#include <sys/types.h>
#include <unistd.h>

void usage(char *arg0)
{
    fprintf(stderr, "Usage: %s count
", arg0);
    exit(EX_USAGE);
}

int main(int argc, char *argv[])
{
    int i;
    int total_points;
    double x;
    double y;
    char *end_ptr;
    ```
if ( argc != 2 )
    usage(argv[0]);

    // Total number of points in the job, 10 in our examples.
    total_points = strtol(argv[1], &end_ptr, 10);

    // Make sure whole argument was a valid integer
    if ( *end_ptr != '\0' )
    {
        fprintf(stderr, "Error: Argument 1 (%s) is not an integer.\n", argv[1]);
        usage(argv[0]);
    }

    if ( total_points < 1 )
    {
        fputs("Total points must be > 0.\n", stderr);
        exit(EX_USAGE);
    }

    // First line of output is the total number of points
    // Not strictly necessary, but provides a level of error-detection
    // in case a file is truncated.
    printf("%d\n", total_points);

    // Initialize pseudo-random number sequence with a different value
    // each time
    srandom((unsigned long)getpid());

    // Compute X random points in the quadrant, and compute how many
    // are inside the circle
    for (i = 0; i < total_points; i++)
    {
        // random() and RAND_MAX are integers, so integer division will
        // occur unless we cast to a real type
        x = (double)random() / RAND_MAX;
        y = (double)random() / RAND_MAX;

        // Print to 20 sig-figs, slightly more than most CPUs can use
        printf("%0.20f %0.20f\n", x, y);
    }

    return EX_OK;

We can generate N input files with a simple script such as the following:

#!/bin/sh
printf "Compiling...\n"
c -O -o gen-points gen-points.c
i=0
while [ $i -lt 10 ]; do
    printf "Generating rand$i.txt...\n"
    ./gen-points 100 > rand$i.txt
    i=$((i+1))
done

The SLURM submission script:
Calcpi-file on an HTCondor Grid

To run a job like this on an HTCondor grid, the trick is to transfer the appropriate input file(s) to each of the execute hosts.

Since we’re compiling the C program on the execute host, we use a shell script as the executable, and list the source file as an input file, along with the actual input file for the program. The shell script is executed on the execute host, where it first compiles the program and then runs it. Below is the job description file:

```bash
#!/bin/sh -e
#SBATCH --output=calcpi-parallel.out
#SBATCH --error=calcpi-parallel.err
#SBATCH --array=1-10
./calcpi-file rand$SLURM_ARRAY_TASK_ID.txt
```
# Pass the process index so that we can use different inputs for each argument
# $(Process) is current job ID.

output = calcpi-parallel.out-$(Process)
error = calcpi-parallel.err-$(Process)

# Logs for the job, produced by HTCondor. This contains output from HTCondor, not from the executable.
log = calcpi-file-condor.log

# Custom job requirements
# HTCondor assumes job requirements from the host submitting job.
# IT DOES NOT DEFAULT TO ACCEPTING ANY ARCH OR OPSYS!!!
# For example, if the job is submitted from peregrine, target.arch is "X86_64" and target.opsys is "FREEBSD8", which do not match POVB execute hosts.
# You can query if your submitting host is accepted by command:
# condor_q -analyze

# Memory requirements in megabytes
request_memory = 50

# Requirements for a binary compiled on CentOS 4 (POVB hosts):
# requirements = (target.arch == "INTEL") && (target.opsys == "LINUX")

# Requirements for a Unix shell script or Unix program compiled on the execute host:
requirements = ((target.arch == "INTEL") || (target.arch == "X86_64")) && ((target.opsys == "FREEBSD") || (target.opsys == "LINUX"))

# Requirements for a job utilizing software installed via FreeBSD ports:
requirements = ((target.arch == "INTEL") || (target.arch == "X86_64")) && ((target.opsys == "FREEBSD")

# Explicitly enable executable transfer mechanism for vanilla universe.
# true | false
transfer_executable = true

# yes | no | if_needed
should_transfer_files = if_needed

# All files to be transferred to the execute hosts in addition to the executable. If compiling on the execute hosts, list the source file(s) here, and put the compile command in the executable script.
transfer_input_files = calcpi-file.c,rand$(Process).txt

# All files to be transferred back from the execute hosts in addition to those listed in "output" and "error".
# transfer_output_files = file1,file2,...

# on_exit | on_exit_or_evict
when_to_transfer_output = on_exit

# Specify how many jobs you would like to submit to the queue.
queue $(Process_count)

The executable, called calcpi-file-condor.sh, would look something like this:

```bash
#!/bin/sh

# Make sure the script was invoked with a filename as a command-line argument
if [ $# != 1 ]; then
   printf "Usage: $0 process-ID\n"
   exit 1
fi

# First command-line argument is the process index
process=$1

# Compile
cc -O -o calcpi-file.$process calcpi-file.c

# Pass the argument to this script on to the C program
./calcpi-file.$process rand$process.txt
```

### 35.3.3 Self-test

1. What is a Monte Carlo experiment?

2. Write a program and a submit script that uses multiple independent processes estimate the probability of rolling a 7 on two dice. This can, or course, be computed directly using simple statistics, but serves as a good exercise for the Monte Carlo method.

   Run the simulation using each of the following parameters. Run with each set of parameters several times, noting the consistency of the results and the computation time for each.

   - One process rolling the dice 100 times.
   - One process rolling the dice 100,000,000 times.
   - Ten processes rolling the dice 1,000,000,000 times each. Make sure each process uses a different random number sequence! (Hint: Use the job array index as a seed.)
Chapter 36

Programming for HPC

Before You Begin
Before reading this chapter, you should be familiar with basic Unix concepts (Chapter 7), the Unix shell (Section 7.4.3, redirection (Section 7.17.1), shell scripting (Chapter 8), and have some experience with computer programming.

36.1 Introduction

There are many computational problems that cannot be decomposed into completely independent subproblems. High Performance Computing refers to a class of distributed parallel computing problems where the processes are not completely independent of each other, but must cooperate in order to solve a problem. The processes within a job are more tightly coupled, i.e. they exchange information with each other while running.

Because HPC processes are not independent of each other, the programming is more complex than HTC. Due to the complexity of HPC programming, it’s usually worth the effort to search for previously written solutions. Most well-known mathematical functions used in science and engineering that can be solved in a distributed parallel fashion have already been implemented. Many complex tasks such as finite element analysis, fluid modeling, common engineering simulations, etc. have also been implemented in both open source and commercial software packages. Chances are that you won’t need to reinvent the wheel in order to utilize HPC.

HPC jobs also may require a high-speed dedicated network to avoid a communication bottleneck. Hence, HPC models are generally restricted to clusters, and few will run effectively on a grid.

HPC problems do not scale as easily as HTC. Generally, more communication between processes means less scalability, but the reality is not so simple. Some HPC models cannot effectively utilize more than a dozen cores. Attempting to use more will increase communication overhead to the point that the job will take as long or longer than it does using fewer cores. On the other hand, some HPC models can scale to hundreds or thousands of cores. Each model is unique and there is no simple way to predict how a model will scale.

36.1.1 Self-test

1. Explain the basic difference between HTC and HPC.
2. What are the advantages of HTC?
3. What are the advantages of HPC?
36.2 Parallelize as a Last Resort

While there will always be a need for parallel computing, the availability of parallel computing resources may tempt people to use them as a substitute for writing good code.

There is virtually no optimal software in existence. Most software in existence at any given moment can be made to run faster, and a significant percentage of it can be made to run orders of magnitude faster. Most performance issues can and should therefore be resolved by optimizing the software first.

Improving software is a more intelligent way to resolve performance issues wherever it’s possible. It will allow effective use of the software on a much wider variety of hardware, possibly including ordinary desktop and laptop machines. This is a much better situation than needing a cluster or grid to get your work done.

It is also the more ethical way to resolve issues where users are running on shared resources. Using tens of thousands of dollars worth of computer equipment to make inefficient software run in a reasonable amount of time is wasteful and foolish, and may delay the work of others who need those resources for more intelligent uses.

Before you consider the use of parallel computing, make sure you’ve done all you can to optimize your software, by choosing efficient algorithms, using compiled languages for the time-consuming parts of your code, and eliminating wasteful code.

Software performance is discussed in greater detail in Part III.

36.3 Common Uses for HPC

One of the most common uses for HPC is modeling fluid flow. This general class of models has a wide range of applications, including weather forecasting, hydraulics, blood flow, etc. When modeling fluid flow, we cannot simply divide the volume into sectors and model flow independently in each sector, since some fluid flows across the boundaries between neighboring sectors. If one process models each sector, each process must exchange some information with the processes modeling neighboring sectors in order to account for all of the fluid in the system.

Another common use for HPC is finite element analysis. Finite element analysis is used to solve partial differential equations, often to model engineering simulations. There are many open source and commercial finite element software packages that utilize HPC.

36.4 Real HPC Examples

Weather Research and Forecasting Model (WRF), http://www.wrf-model.org/index.php, is an HPC software package commonly used in atmospheric science research.

Finite Volume Coastal Ocean Model (FVCOM), http://fvcom.smast.umassd.edu/FVCOM/index.html is an HPC software package used to predict ocean currents in a near-shore environment.

Finite element software packages are too numerous to list here. For a current listing, see the Wikipedia article on finite element analysis software http://en.wikipedia.org/wiki/List_of_finite_element_software_packages.

36.5 HPC Programming Languages and Libraries

HPC programming can be done using a wide variety of languages and tools, but most major HPC software is written in C, C++, or Fortran using OpenMP, POSIX Threads, or the Message Passing Interface (MPI) libraries.

C and Fortran are pure compiled languages, so programs run at the highest possible speed (often orders of magnitude faster than the same program written in an interpreted language). C++ also offers good performance, but C++ has a higher learning curve and best suited for well-trained, experienced programmers.

When developing libraries to be used from C, C++, and Fortran programs, C is generally the most trouble-free choice. C libraries are very easy to link into C++ and Fortran programs, and generally provide marginally better performance than C++ or Fortran. C++ and Fortran libraries can also be linked into programs written in other languages, but the process requires additional knowledge and effort.
36.5.1 Self-test

1. What are the most popular languages for HPC programming? Why?

36.6 Shared Memory Parallel Programming with OpenMP

OpenMP, short for Open Multiprocessing, is a set of tools embedded into many Fortran compilers since 1997, and C/C++ compilers since 2000. OpenMP allows the programmer to utilize multiple cores within a single computer, and in some cases, using extensions, multiple computers.

OpenMP allows for finer grained parallelism than embarrassingly parallel computing or distributed models such as MPI.

The latter two paradigms utilize multiple processes, potentially running on different computers, which require a great deal of overhead to create. Hence, individual processes need to be long-running so that the overhead of starting them doesn’t use a significant portion of the total run time.

OpenMP, on the other hand, spawns lightweight threads within a single process, which requires very little overhead. Hence, OpenMP can effectively parallelize very short-running sections of code such as individual loops within a program.

OpenMP can benefit code that might only take a fraction of a second to run in serial, whereas HTC and MPI are only good for parallelizing processes that take far longer (usually minutes or hours).

The main limitation of OpenMP is that it only scales up to the number of cores available on a single computer (or other tightly-coupled architecture). If you want to achieve greater speedup, you’ll need to look for ways to perform coarser scale parallelism with HTC or MPI.

OpenMP is implemented as a set of preprocessor directives, header files, and library functions.

To use OpenMP, a C or C++ program must include the header `omp.h` and use `#pragma` directives to indicate which code segments are to be parallelized and how.

Fortran programs must contain the line

```
use omp_lib
```

and use specially formatted comments beginning with

```
!$omp
```

To compile an OpenMP program with gcc, we must use the `-fopenmp` flag:

```
mpyc: gcc -fopenmp myprog.c
mpyc: g++ -fopenmp myprog.cc
mpyc: gfortran -fopenmp myprog.f90
```

To compile an OpenMP program with clang, we need clang 3.7 or later and must use the `-fopenmp` compile flag and the `-lomp` link flag. For example, on a FreeBSD system with the devel/clang37 port installed:

```
mpyc: clang37 -fopenmp myprog.c -L/usr/local/lib -lomp
mpyc: clang++37 -fopenmp myprog.cc -L/usr/local/lib -lomp
```

At the time of this writing, clang does not include a Fortran compiler. Clang is intended to be binary compatible with GCC, however, so we can use gfortran alongside clang and clang++ as long as the compiler versions are compatible.

C and C++ programs must also contain

```
#include <omp.h>
```

After an openmp program has split into multiple threads, each thread can identify itself by calling `omp_get_thread_num()`, which returns a value between 0 and N-1 for a program running N threads.

Multithreaded programs can be `task parallel` (running different code for each thread) or `data parallel` (running the same code on different data for each thread).
36.6.1 OMP Parallel

Using a basic omp parallel pragma, we can execute the same code on multiple cores (more or less) simultaneously.

Simple parallel program:

```c
#include <stdio.h>
#include <sysexits.h>
#include <omp.h>

int main(int argc, char *argv[])
{
    /* Execute the same code simultaneously on multiple cores */
    #pragma omp parallel
        printf("Hello from thread %d\n", omp_get_thread_num());
    #pragma omp end parallel
    return EX_OK;
}
```

```fortran
program openmp_hello
    use omp_lib

    ! Disable implicit declarations (i-n rule)
    implicit none

    ! Note: No space allowed between ! and $
    !$omp parallel
    !$omp print *, 'Hello from thread ', omp_get_thread_num()
    !$omp end parallel
end program
```
36.6.2 OMP Loops

One of the most commonly used features of OpenMP is loop parallelization.

Note As always, you should optimize the serial code before trying to parallelize it. Before resorting to parallelizing a loop, make sure that the loop is as efficient as possible. You may be able to move some code outside the loop to reduce its run time, greatly reduce the number of iterations, or eliminate the loop entirely with some careful thought. See Section 23.15 for a detailed discussion.

If a loop can be structured in such a way that each iteration is independent of previous iterations, then in theory, all of the iterations can be executed at the same time.

Of course, the number of iterations which can execute at once is limited by the number of cores available. For example, if a parallelizable loop iterates 100 times, but the computer running it only has four cores, then only four iterations will run at once. Still, this will speed up the program by nearly a factor of four. A small amount of overhead is incurred when splitting the execution path into multiple threads and again when merging them back together after the parallel segment.

Example of a parallel loop with OpenMP:

```c
#include <stdio.h>
#include <omp.h>
#include <sysexits.h>

int main(int argc, char **argv)
{
    int c;

    #pragma omp parallel for
    for (c=0; c < 8; ++c)
    {
        printf("Hello from thread %d, nthreads %d, c = %d\n",
            omp_get_thread_num(), omp_get_num_threads(), c);
    }
    return EX_OK;
}
```

Example of a parallel loop with OpenMP:

```c
#include <stdio.h>
#include <omp.h>
#include <sysexits.h>

int main(int argc, char **argv)
{
    int c;

    #pragma omp parallel for
    for (c=0; c < 8; ++c)
    {
        printf("Hello from thread %d, nthreads %d, c = %d\n",
            omp_get_thread_num(), omp_get_num_threads(), c);
    }
    return EX_OK;
}
```
! Main program body
program openmp_hello
  use omp_lib

  ! Disable implicit declarations (i-n rule)
  implicit none
  integer :: c

  ! Note: No space allowed between ! and $
  !$omp parallel do
do c = 1, 8
    print *, 'Hello from thread ', omp_get_thread_num(), &
    ' nthreads ', omp_get_num_threads(), ' c = ', c
  enddo
end program

---

**Caution**

When several iterations of a loop are run in parallel, we cannot predict the order in which they will complete. While they are in theory executed at the "same time", there really is no such thing as exactly the same time. Even if each iteration runs the exact same number of instructions, they may finish at slightly different times, and the order is unpredictable. Hence, when using OpenMP, you must take special care to ensure that it does not matter to your program which iterations complete first.

Performing output to a common stream such as the standard output in a parallelized section of code is generally a bad idea, except for debugging purposes.

---

If anything within a loop depends on results computed during previous iterations, then the loop simply cannot be parallelized. It may be possible to redesign the loop so that each iteration is independent of the rest, but there are some cases where computations must be done serially.

### 36.6.3 Shared and Private Variables

When an OpenMP process splits into multiple threads, we may want some variables to exist independently in each thread and others to be shared by all threads.

For example, if a parallelized loop computes the sum of a list of numbers, the sum variable must be shared.

On the other hand, any variable that must contain a different value for each iteration (such as the OMP thread number) should be private.

### 36.6.4 Critical and Atomic Sections

When multiple threads modify a shared variable, there is a danger that the modifications could overlap, resulting in incorrect results.

---

**Caution** What appears as one statement in the source code may actually be a sequence of several instructions of machine code.
For example, when assigning a new value to sum in the example below, a typical CPU must actually perform multiple steps. The following would represent the sequence of machine instructions on many CPUs:

1. Load the value of sum from memory into the CPU
2. Load the value of c_squared from memory into the CPU
3. Add sum + c_squared
4. Store the result back to memory variable sum

```c
#include <stdio.h>
#include <sysexits.h>

int main(int argc,char *argv[])
{
    int c, sum, c_squared;

    sum=0;
    /* compute sum of squares */
#pragma omp parallel for private(c_squared) shared(sum)
    for (c=0; c<20; ++c)
    {
        c_squared = c * c;
#pragma omp atomic
        sum += c_squared;
    }
    printf("%d\n", sum);
    return EX_OK;
}
```

!-----------------------------------------------------------------------
! Program description:  
! OpenMP parallel loop example
!-----------------------------------------------------------------------

! Modification history:  
! Date Name Modification  
! 2012-06-29 Jason Bacon Created
!-----------------------------------------------------------------------

! Main program body
program openmp_hello
```
Suppose there are two threads, numbered 0 and 1.

Now suppose thread 0 begins the sequence when sum = 1 and c = 2. When thread 0 finishes the sequence, sum will contain 5. If thread 1 begins the sequence immediately after thread 0 finishes, it will add 5 + 9 and store the correct value of 14 in sum.

However, if thread 1 begins before thread 0 stores the 5 into sum, the following will occur:

1. Thread 1 will load the value 1 from sum, not 5.
2. Thread 0 will store 5 into sum as expected, but thread 1 will never load this value because it as already performed its load instruction.
3. Thread 1 will then add 1 + 9 and store the incorrect value 10 in sum.

Unfortunately, the programmer must take responsibility for pointing this out to the compiler by declaring statements as critical or atomic.

A statement or block marked critical can only be entered (begun) by one thread at a time. Once a thread has begun executing a critical section, no other thread can begin executing it until the current thread has completed it.

In an atomic section, only memory update portions of the code are considered critical. This portion would include reading the old value of sum and writing the new value in the example above. Reading the value of c_squared would not be critical, and could be executed concurrently by multiple threads. Using atomic is usually sufficient, and sometimes faster, since it may allow the compiler to parallelize parts of the machine code that cannot be separated from the critical parts at the source code level.

36.6.5 Self-test

1. What are the advantages of OpenMP over distributed parallel systems such as MPI?

2. Write an OpenMP program that prints the square of every number from 1 to 1000. Print the thread number alongside each square.

36.7 Shared Memory Parallelism with POSIX Threads

The POSIX threads library, or pthreads, is a standardized interface for using lightweight threads in C programs on POSIX (Portable Operating System Interface based on Unix) platforms.

Note that C libraries can easily be used in programs written in other compiled languages, so pthreads can be used directly in C++ and Fortran programs. There are also well-developed interfaces for many other languages.

The pthreads system addresses the same basic needs as OpenMP, but using a different approach. Threads are created manually using pthread_create() in much the same way as we would use fork() to create a new process.
/*
 * Example from Pthreads tutorial at:
 * https://computing.llnl.gov/tutorials/pthreads/
 */

#include <pthread.h>
#include <stdio.h>
#define NUM_THREADS 5

void *PrintHello(void *threadid)
{
    long tid;
    tid = (long)threadid;
    printf("Hello World! It’s me, thread %ld\n", tid);
    pthread_exit(NULL);
}

int main (int argc, char *argv[])
{
    pthread_t threads[NUM_THREADS];
    int rc;
    long t;
    for(t=0; t<NUM_THREADS; t++){
        printf("In main: creating thread %ld\n", t);
        rc = pthread_create(&threads[t], NULL, PrintHello, (void *)t);
        if (rc)
        {
          printf("ERROR; return code from pthread_create() is %d\n", rc);
          exit(-1);
        }
    }

    /* Last thing that main() should do */
    pthread_exit(NULL);
}

36.8 Message Passing Interface (MPI)

Avoid it if you can. Embrace it wholeheartedly if you must...

The Message Passing Interface (MPI) is a standard API (application program interface) and set of tools for building and running distributed parallel programs on almost any parallel computing architecture.

MPI includes libraries of subprograms that make it as simple as possible to start up a group of cooperating processes and pass messages between the processes. It also includes tools for running, monitoring, and debugging MPI jobs.

There are many implementations of MPI, but the emerging standard is Open MPI, which evolved from the best features of several earlier open source implementations. Open MPI is free and open source, so you can rest assured that projects you develop with open MPI will never be orphaned.

It is important to note that MPI programs do not require a cluster to run. MPI is also effective in taking advantage of multiple cores in a single computer. MPI programs can even be run on a single-core computer for testing purposes, although this won’t, of course, run any faster than a serial program on the same machine, and may even take slightly longer. Other parallel programming paradigms might be easier to use or faster than MPI on a shared memory architecture, but if you may want to run across multiple computers, programming in MPI is a good investment of time.

The bottom line is that you can use the same MPI programs to utilize multiple cores on a single computer, or a cluster of any size suitable for the job. This makes MPI the most portable applications programming interface (API) for parallel programs. You
can also develop and test MPI code on your own PC, and later run it on a larger cluster. Some users may find this approach preferable, since development on a local PC with their preferred tools can be faster and more comfortable, and it reduces the risk of impacting other cluster users with untested code.

36.8.1 Self-test

1. What does MPI stand for?
2. What tools does the MPI system include?
3. What type of parallel computer system does MPI require?

36.9 MPI vs Other Parallel Paradigms

There are a number of ways to decompose many problems for parallel execution.

If a problem can be decomposed into independent parts and run in an embarrassingly parallel fashion, this is usually the best route, since it is the easiest to program and the most scalable.

If the processes within a parallel job must communicate during execution, there are a variety of options, including MPI, shared-memory parallelism, and specialty hardware architectures (supercomputers) such as Single Instruction Multiple Data (SIMD) machines.

Which architecture and model will provide the best performance is dependent on the algorithms your software needs to use.

One advantage of MPI, however, is portability. An MPI program is capable of utilizing virtually any architecture with multiple processors. MPI can utilize the multiple cores on a single PC, the multiple nodes in a cluster, and in some cases (if communication needs are light) MPI programs could even run on a grid.

Shared-memory parallelism, discussed in Section 36.6, might provide better performance when utilizing multiple cores within a single PC. However, this is not a general rule. Again, it depends on the algorithms being used. In addition, shared-memory parallelism does not scale well, due to the fact that the cores contend for the same shared memory banks and other hardware resources. A PC with 48 cores may not provide the performance boost you were hoping for.

Since parallel programming is a complex and time-consuming endeavor, using a system such as MPI that will allow the code to run on the widest possible variety of architectures has clear advantages. Even if shared-memory parallelism offers better performance, it may still be preferable to use MPI when you consider the value of programmer time and the ability to utilize more than one computer. The performance gains of a shared-memory program using a small number of cores may not be important enough to warrant the extra programming time and effort.

36.9.1 Self-test

1. When should embarrassingly parallel computing be used instead of parallel programming?
2. What are the general rules that indicate the best parallel programming paradigm for a given problem?
3. What are the limitations of shared-memory parallelism?

36.10 Structure of an MPI Job

An MPI job consists of two or more cooperating processes which may be running on the same computer or on different computers. MPI programs can be compiled with a standard compiler and the appropriate compile and link flags. However, MPI systems provide simple wrappers that eliminate the need to include the MPI flags.

For example, an MPI program can be run using 4 cores on a stand-alone computer with:
One process in the job is generally designated as the root process. The root process is not required to do anything special, but it typically plays a different role than the rest. Often the root process is responsible for things like partitioning a matrix and distributing it to the other processes, and then gathering the results from the other processes.

Usually, mpirun starts up N identical processes, which then determine for themselves which process they are (by calling an MPI function that returns a different rank value to each process) and then follow different paths depending on the result.

```c
/* Initialize data for the MPI functions */
if ( MPI_Init(&argc, &argv) != MPI_SUCCESS )
{
  fputs("MPI_Init failed.
", stderr);
  exit(EX_UNAVAILABLE);
}
if ( MPI_Comm_rank(MPI_COMM_WORLD, &my_rank) != MPI_SUCCESS )
{
  fputs("MPI_Comm_rank failed.
", stderr);
  exit(EX_UNAVAILABLE);
}
/*
 * For this job, the process with rank 0 will assume the role of
 * the "root" process, which will run different code than the
 * other processes.
 */
if (my_rank == ROOT_RANK)
{
  // Root process code
}
else
{
  // Non-root process code
}
```

Note All processes in this MPI job contain code that will never be executed, but this is not considered a problem, since the size of code is generally dwarfed by the size of the data. Hence, it is not usually worth the effort to create separate programs for different processes within an MPI job.

## 36.11 A Simple MPI Program

MPI programming is complex and not necessary for all cluster users. Many cluster users will only run HTC jobs, which do not require communication between processes. The goal of this example is to provide a very simple introduction to MPI programming for those who are interested. We assume some familiarity with C or Fortran.

Users interested in pursuing MPI programming are encouraged to consult a book on the subject. Many good resources are cited on the HPC website at [http://www4.uwm.edu/hpc/related_resources/](http://www4.uwm.edu/hpc/related_resources/).

C version:

```c
/*
 * Program description:
 *
 * MPI Example
 */
```
A typical MPI job will start N processes, all running the same program. This is known as the Single-Program Multiple-Data (SPMD) model.

This program demonstrates how the various processes distinguish themselves from the rest using MPI library functions.

Return values of all MPI functions are checked for good practice, although some MPI functions will by default terminate the process without returning if an error is encountered. This behavior can be changed, so including error checks for every MPI call is a good idea.

```c
#include <stdio.h> /* fputs(), printf(), etc. */
#include <stdlib.h> /* exit() */
#include <unistd.h> /* gethostname() */
#include <mpi.h> /* MPI functions and constants */

#define ROOT_RANK 0
#define MESSAGE_MAX_LEN 128
#define TAG 0

int main(int argc, char *argv[]) {
    int total_processes;
    int my_rank;
    int rank;
    MPI_Status status;
    char message[MESSAGE_MAX_LEN + 1];
    char hostname[MAXHOSTNAMELEN + 1];

    /* Get name of node running this process */
    if ( gethostname(hostname, MAXHOSTNAMELEN) != 0 ) {
        fputs("gethostname() failed.\n", stderr);
        exit(EX_OSERR);
    }

    /* Initialize data for the MPI functions */
    if ( MPI_Init(&argc, &argv) != MPI_SUCCESS ) {
        fputs("MPI_Init failed.\n", stderr);
        exit(EX_UNAVAILABLE);
    }

    /* Find out how many processes are in this MPI job */
    if ( MPI_Comm_size(MPI_COMM_WORLD, &total_processes) != MPI_SUCCESS ) {
        fputs("MPI_Comm_size failed.\n", stderr);
        exit(EX_UNAVAILABLE);
    }

    return TAG;
}
```
* Each process within the job has a unique integer "rank".
* This is how each process determines its role within the job.
*/
if ( MPI_Comm_rank(MPI_COMM_WORLD, &my_rank) != MPI_SUCCESS )
{
  fputs("MPI_Comm_rank failed.
", stderr);
  exit(EX_UNAVAILABLE);
}

/*
* For this job, the process with rank 0 will assume the role of
* the "root" process, which will run different code than the
* other processes.
*/
if (my_rank == ROOT_RANK)
{
  printf("We have %d processors\n", total_processes);
  /* Send a message to all non-root processes */
  for (rank = 1; rank < total_processes; ++rank)
  {
    snprintf(message, MESSAGE_MAX_LEN, "Process %d, where are you? ", rank);
    if ( MPI_Send(message, MESSAGE_MAX_LEN, MPI_CHAR, rank, TAG,
             MPI_COMM_WORLD) != MPI_SUCCESS )
    {
      fputs("MPI_Comm_rank failed.
", stderr);
      exit(EX_UNAVAILABLE);
    }
  }
  /* Read the response from all non-root processes */
  for (rank = 1; rank < total_processes; ++rank)
  {
    if ( MPI_Recv(message, MESSAGE_MAX_LEN, MPI_CHAR, rank, TAG,
              MPI_COMM_WORLD, &status) != MPI_SUCCESS )
    {
      fputs("MPI_Comm_rank failed.
", stderr);
      exit(EX_UNAVAILABLE);
    }
    printf("%s\n", message);
  }
}
else
{
  /* Wait for message from root process */
  if ( MPI_Recv(message, MESSAGE_MAX_LEN, MPI_CHAR, ROOT_RANK,
            MPI_COMM_WORLD, &status) != MPI_SUCCESS )
  {
    fputs("MPI_Comm_rank failed.
", stderr);
    exit(EX_UNAVAILABLE);
  }
  printf("Process %d received message: %s\n", my_rank, message);

  /* Send response */
  snprintf(message, MESSAGE_MAX_LEN, "Process %d is on %s", my_rank, hostname);
  if ( MPI_Send(message, MESSAGE_MAX_LEN, MPI_CHAR, ROOT_RANK, TAG, MPI_COMM_WORLD) != MPI_SUCCESS )
  {
    fputs("MPI_Comm_rank failed.
", stderr);
    exit(EX_UNAVAILABLE);
  }
}
/*
 * All MPI processes must execute MPI finalize to synchronize
 * the job before they exit.
 */

if ( MPI_Finalize() != MPI_SUCCESS )
{
    fputs("MPI_Finalize failed.\n", stderr);
    exit(EX_UNAVAILABLE);
}
return EX_OK;

Fortran version:

!-----------------------------------------------------------------------
! Program description:
!  MPI Example
!  A typical MPI job will start N processes, all running the same
!  program. This is known as the Single-Program Multiple-Data (SPMD)
!  model.
!  This program demonstrates how the various processes
!  distinguish themselves from the rest using MPI library functions.
!-----------------------------------------------------------------------

module constants
  ! Global Constants
  double precision, parameter ::
     PI = 3.1415926535897932d0, E = 2.7182818284590452d0,
     TOLERANCE = 0.000000000001d0, AVOGADRO = 6.0221415d23
     ! For numerical methods
  integer, parameter ::
     MESSAGE_MAX_LEN = 128, HOSTNAME_MAX_LEN = 128, TAG = 0,
     ROOT_RANK = 0
end module constants

! Main program body
program mpi_hello
  use constants     ! Constants defined above
  use ISO_FORTRAN_ENV  ! INPUT_UNIT, OUTPUT_UNIT, ERROR_UNIT, etc.

  ! Disable implicit declarations (i-n rule)
  implicit none

  include 'mpif.h'     ! MPI constants

  ! Variable definitions
  character(MESSAGE_MAX_LEN) :: message     ! MPI message buffer
character(HOSTNAME_MAX_LEN) :: hostname ! Name of node
integer :: total_processes, my_rank, rank, count, ierr, &
  message_length = MESSAGE_MAX_LEN
integer :: status(MPI_STATUS_SIZE)

! Get name of node running this process
call hostnm(hostname)

! Initialize data for the MPI functions
call mpi_init(ierr)
if ( ierr /= MPI_SUCCESS ) stop 'mpi_init failed.'

! Find out how many processes are in this MPI job
call mpi_comm_size(MPI_COMM_WORLD, total_processes, ierr)
if ( ierr /= MPI_SUCCESS ) stop 'mpi_comm_size failed.'

! Each process within the job has a unique integer "rank".
! This is how each process determines its role within the job.
call mpi_comm_rank(MPI_COMM_WORLD, my_rank, ierr)
if ( ierr /= MPI_SUCCESS ) stop 'mpi_comm_rank failed.'

! For this job, the process with rank ROOT_RANK will assume the role of
! the "root" process, which will run different code than the
! other processes.
if ( my_rank == ROOT_RANK ) then
  ! Only root process runs this clause
  ! Do this in root so it only prints once
  print '(a,i0,a)', 'We have ', total_processes, ' processes.'

  ! Debug code
  ! print '(a,a,a)', 'Root processing running on ', trim(hostname), '.'
  do rank = 1, total_processes-1
    write (message, '(a,i0,a)') 'Process ', rank, ', where are you?'
    ! Debug code
    ! print '(a,a,a,i0,a)', 'Sending ', trim(message), ' to process ', &
    ! rank, '. '
    ! It’s stupid to send a padded string, but it’s complicated
    ! for mpi_recv() to receive a message of unknown length
    ! A smarter program would save network bandwidth and time by using
    ! len_trim(message) instead of MESSAGE_MAX_LEN.
call mpi_send(message, MESSAGE_MAX_LEN, MPI_CHARACTER, &
  MPI_ANY_SOURCE, MPI_ANY_TAG, MPI_COMM_WORLD, status, ierr)
  if ( ierr /= MPI_SUCCESS ) stop 'mpi_send failed.'
  enddo

  do count = 1, total_processes-1
    ! Accept message from slave processes in any rank order
    ! by using MPI_ANY_SOURCE for rank in recv call
    call mpi_recv(message, MESSAGE_MAX_LEN, MPI_CHARACTER, &
      MPI_ANY_SOURCE, MPI_ANY_TAG, &
      MPI_COMM_WORLD, status, ierr)
    if ( ierr /= MPI_SUCCESS ) stop 'mpi_recv failed.'
    print *, 'Root received response: ', trim(message)
  enddo
else
  ! All slave processes run this section
  ! Debug code
In a scheduled environment, MPI jobs are submitted like batch serial jobs. The scheduler is informed about resource requirements (cores, memory) but does not dispatch all the processes. The scheduler dispatches a single `mpirun` command and the `mpirun` command then creates all the processes to a list of nodes provided by the scheduler.

**SLURM submit script:**

```bash
#!/bin/sh -e
# Number of cores
#SBATCH --ntasks=8
mpirun ./mpi-hello
```

**PBS submit script:**

```bash
#!/bin/sh
# Job name
#PBS -N MPI-Hello
#SBATCH --ntasks=8
```

Programs should be compiled in the same environment in which they will run, i.e. on a compute node, under the scheduler. This will ensure that they find the same tools and libraries at run time as they did at compile time. The best way to achieve this is by using a submit script to compile:

**SLURM build script:**

```bash
#!/bin/sh -e
# Number of cores
#SBATCH --ntasks=1
mpicc -o mpi-hello mpi-hello.c
```

**PBS build script:**

```bash
#!/bin/sh
# Number of cores
#PBS -l procs=8
```
# Shell commands

# Torque starts from the home directory on each node, so we must manually
# cd to the working directory where the hello binary is located.

cd $PBS_O_WORKDIR
mpirun ./mpi-hello

LSF submit script:

#!/usr/bin/env bash

# Job name
#BSUB -J MPI-Hello

# Number of cores
#BSUB -n 8

# LSF requires the use of wrapper scripts rather than using mpirun directly
openmpi_wrapper ./mpi-hello

### 36.11.1 Self-test

1. Write an MPI version of calcpi.c from Chapter 35. For simplicity, use a constant for the number of random points generated and use the process rank for the srandom() seed.

2. Is MPI the best solution for estimating PI? Why or why not?

3. Write an MPI matrix addition routine. The root process should distribute a row of each source matrix to each of the worker processes until all the rows are added.

4. Is this a good fit for MPI? Why or why not?

### 36.12 Best Practices with MPI

The MPI system attempts to clean up failed jobs by terminating all processes in the event that any one of the processes fails. However, MPI’s automatic cleanup can take time, and cannot always detect failures. This sometimes leads to orphaned processes remaining in the system unbeknownst to the scheduler, which can cause problems for other jobs.

Hence, it is each programmer’s responsibility to make sure that their MPI programs do not leave orphaned processes hanging around on the cluster. How this is accomplished depends on the particular program, however every MPI program should follow these general rules:

1. Check the exit status of every MPI function/subroutine call and every other statement in the program that could fail in a way that would prevent the program from running successfully. Some other examples include memory allocations and file opens, reads, writes. These are only examples, however. It is the programmer’s responsibility to examine every line of code and check for possible failures. **There should be no exceptions to this rule.**

2. Whenever a statement fails, the program should detect the failure and take appropriate action. If you cannot determine a course of action that would allow the program to continue, then simply perform any necessary cleanup work and terminate the process immediately. MPI programs that do not self-terminate may leave orphaned processes running on a cluster that interfere with other users’ jobs.
36.12.1 Self-test

1. How can an MPI program ensure that it doesn’t leave orphaned processes running?

36.13 Higher Level MPI Features

A parallel program may not provide much benefit if only the computations are done in parallel. Disk I/O and message passing may also need be parallelized in order to avoid bottlenecks that will limit performance to serial speed.

36.13.1 Parallel Message Passing

Suppose we have 100 processes in an MPI job that all need the same data in order to begin their calculations. We could simply loop through them and send the data as follows:

```c
for (rank = 1; rank < 100; ++rank)
{
    if ( MPI_Send(data, len, MPI_CHAR, rank, TAG, MPI_COMM_WORLD) != MPI_SUCCESS )
    {
        ...
    }
}
```

The problem is, this is a serial operation that sends one message at a time, while many pathways through the network switch may be left idle. A typical network switch used in a cluster is capable of transmitting many messages between disjoint node pairs at the same time. For example, node 1 can send a message to node 5 at the same time node 4 sends a message to node 10.

If it ends up taking longer to distribute data to the processes than it does to do the computations on that data, then it’s time to look for a different strategy.

A simple strategy to better utilize the network hardware might work as follows:

1. Root process transmits data to process 1.
2. Root process and process 1 transmit to processes 2 and 3 at the same time.
3. Root process and processes 1, 2, and 3 can all transmit to processes 4, 5, 6, and 7 at the same time.
4. ...and so on.

If the job uses a large number of processes, the limits of the network switch may be reached, but that’s OK. If the switch is saturated, it will simply transmit the data as fast as it can. There is rarely any negative impact from a saturated network switch, other than increased response times for other processes, but this is why clusters have dedicated networks.

While this broadcast strategy is simple in concept, it can be tricky to program. Real world strategies take into account various different network architectures in an attempt to optimize throughput for specific hardware.

Fortunately, MPI offers a number of high-level routines such as MPI_Bcast(), MPI_Scatter(), and MPI_Gather(), which will provide good performance on most hardware, and certainly better than the serial loop above.

There may be cases where using routines designed for specific network switches can offer significantly better performance. However, this may mean sacrificing portability.

A big part of becoming a proficient MPI programmer is simply learning what MPI has to offer and choosing the right routines for your needs.
36.13.2 Self-test

1. What is the easiest way to ensure reasonably good performance from your MPI programs?

2. What problems are associated with overloading a cluster’s dedicated network?

3. Write an MPI program that solves a linear system of equations using an open source distributed parallel linear systems library.

36.14 Process Distribution

Another thing we need to consider on modern hardware is that most cluster nodes have multiple cores. Hence, messages are often passed between two processes running on the same compute node. Such messages do not pass through the cluster network: There is no reason for them to leave the compute node and come back. Instead, they are simply placed in an operating system memory buffer by the sending process, and then read by the recipient process.

In general, local connections like this are faster than even the fastest of networks. However, memory is a shared resource and can become a bottleneck if too many processes on the same node are passing many messages.

For some applications, you may find that you get better performance by limiting the number of processes per node, and hence balancing the message passing load between local memory and the network. Jobs that perform a lot of disk I/O may also benefit from using fewer processes per node. Most schedulers allow you to specify not just how many processes to use, but also how many to place on each node.

The best way to determine the optimal distribution of processes is by experimenting. You might try 1 per node, 4 per node, and 8 per node and compare the run times. If 8 per node turns out to be the slowest, then try something between 1 and 4. If 4 turns out to be the fastest, then try 5 or 6. Results will vary on different clusters, so you’ll have to repeat the experiment on each cluster. If you plan to run many jobs over the course of weeks or months, spending a day or two finding the optimal distribution could lead to huge time savings overall.

36.14.1 Self-test

1. When is it likely that limiting the number of MPI processes per node will improve performance? Why?

36.15 Parallel Disk I/O

Disk I/O is another potential bottleneck that can kill the performance of a parallel program. Disk access is about 1,000,000 times slower than memory access in the worst case (completely random), and about 10 or 20 times slower in the best case (completely sequential).

I/O is generally avoided even in serial programs, but the problem can be far worse for parallel programs. Imagine 300 processes in a parallel job competing for access to the same files.

Using local disks on the compute nodes might improve performance, especially if the processes are spread out across as many nodes as possible. However, this will require distributing the data beforehand. Another issue is that modern nodes have multiple cores, and there may be other processes competing for disk even though you chose to spread out your own job. You could request exclusive access to nodes if I/O is really an issue, but leaving 7 cores idle on 300 8-core machines is not good utilization of resources.

Modern clusters usually have a high-speed shared filesystem utilizing Redundant Arrays of Parallel Disks (RAIDs). In theory, a group of 20 disks grouped together in a RAID can be read 20 times faster than a single disk of the same type. Performance gains for write access are not as simple, but can also be significant.

Depending on how much I/O your jobs must do, you may be better off using a high-speed shared RAID, or you may be better off distributing data to local disks before computations begin. Only experimentation with each job on each cluster will reveal the optimal strategy.
Part V

Systems Management
Chapter 37

Platform Selection

TBD
Chapter 38

System Security

• Configure firewall or TCP wrappers to allow incoming traffic from only specific hosts.

• Create ONE account with administrator rights and use it only for system updates and software installations.

• Do not share login accounts. Create SEPARATE accounts for each user, without administrator rights, and use them for all normal work.

• NEVER share your password with ANYONE. PERIOD. NOBODY should ever ask you for your password. Other users have no right to mess with your login account. IT staff with rights to manage a machine do not need your password, so be suspicious if they ask for it.

• Store passwords in KeePassX or a similar encrypted password vault. Use a strong password for each KeePassX database.

• If you set up a computer to allow remote access, use ONLY systems that encrypt ALL traffic. If you are not sure your remote access software encrypts everything, DO NOT ENABLE IT. Talk to a professional about how to securely access the computer remotely before allowing it.

• If you suspect that your computer has been hacked, unplug it from the network (or disable WiFi), but do not turn it off. Call your local computer security experts, and do not touch the computer until they arrive.
Chapter 39

Index

A
alpha testing, 233

B
beta testing, 233

C
central processing unit, 154
core, 154
Cray, 154
CUDA, 156

D
distributed parallelism, 155
distributed processing, 155

E
embarrassingly parallel, 154, 156

G
gpu, 156
graphics processing unit, 156
grid, 156

H
high performance computing, 155
HPC, 155
HTCondor, 156

I
integration testing, 233
interface, 338

L
Load Sharing Facility, 203
LSF, 203

N
node, 155

O
object-oriented, 222

P
packages system, 99
Platform Cluster Management, 203
Pools Of Virtual Boxes, 156
ports system, 99
POVB, 156
processor, 154

R
rsync, 84

S
scale, 155
signature, 338
SIMD, 157
single-instruction, multiple-data, 157
software life cycle, 232
ssh_config, 102
stub, 338
supercomputer, 157

testable stub, 338
thread, 155

U
UI, 36
User interface, 36

V
vector processor, 157
video processor, 156
virtual desktop, 38

W
workspace, 38