The CSC 214 Guide: Programming Concepts in C++

Suzanne Balik  
spbalik@unity.ncsu.edu

Matthias Stallmann  
mfms@csc.ncsu.edu

July 3, 2000
Acknowledgements

We would like to thank all the CSC 210/214 instructors and TA’s who had a role in producing this document, especially Dan DuVarney, Mike Lake, Carol Miller, and Jo Perry.

This document was typeset using LaTeX [Lam94].

Other C++ books that were especially helpful in its preparation (some were previously used as CSC 210 texts) include Perry and Levin [PL96], Headington and Riley [HR94], Deitel and Deitel [DD98], Eckel [Eck95], Astrachan [Ast97], Adams, Leestma, and Nyhoff [ALN95], and Sta- ugaard [Sta97]. Books about programming language concepts, such as Marcotty and Ledgard [ML86] have also been helpful as a way of organizing the information and understanding how C++ relates to other languages. The C++ language was designed by Bjarne Stroustrup [Str91, ES90].

Disclaimer

This is the first time all concepts traditionally taught in CSC 210/214 have been brought together in one document. Like all documents, it is a snapshot of what we were able to write and edit by a particular date. But the process of gathering and refining this material continues. We hope that you find much useful information and incentive for further learning in these pages (the bibliography at the end should serve as a guide).

We also hope that you will not be shy about pointing out the omissions and errors that are sure to be present, and making suggestions for improving the presentation.

Third Edition

We thank the following students who have pointed out errors and confusions in previous editions and helped us make this third edition a major improvement over the previous ones: Stan Converse, Nianci Gan, Fang He, David Heslop, Karen Kennedy, Ken Krebs, Jordan Liggett, Kristin Mastin, Sherrie Potter-Baldwin, Lucy Whitehead, and Ray Zill.
# Contents

1 Software Design .................................................. 11
   1.1 Algorithms and Problem Solving .......................... 11
   1.2 Top-Down Design and Stepwise Refinement .............. 12
   1.3 Object-Oriented Design .................................. 13
   1.4 The “Waterfall” Model ................................ 17
   1.5 Verification: Logical and Experimental ................. 21
   1.6 Debugging .................................................. 24
   1.7 Key Concepts ............................................. 31
   1.8 Notes ....................................................... 34

2 Separate Compilation and Makefiles ............................. 35
   2.1 Why bother? .............................................. 39
   2.2 Header files ............................................ 40
   2.3 Makefiles ................................................ 41
   2.4 Makefile errors ........................................ 44
   2.5 Other features of makefiles ............................... 45
   2.6 A “GNU” syntax for makefiles ........................... 46
   2.7 Compiler options ....................................... 47
   2.8 Key Concepts ......................................... 47

3 Finite-State Machines ........................................... 49
   3.1 Text Processing with FSM’s ............................. 51
   3.2 Horner’s Rule .......................................... 56
   3.3 Finite-state machine based on Horner’s Rule .......... 57
   3.4 Key Concepts .......................................... 57

4 Classes .............................................................. 61
   4.1 Background .............................................. 61
   4.2 The Date class .......................................... 61
   4.3 The DateList class ..................................... 66
   4.4 Structures vs. Classes ................................ 71
   4.5 Key Concepts .......................................... 73
5 Pointers and Dynamic Memory Management 75
  5.1 What are Pointers? .................................................. 75
  5.2 Pointers in C++: * and & ....................................... 75
  5.3 Aliases .................................................................. 77
  5.4 Dynamic Memory ..................................................... 77
  5.5 Heap Manager .......................................................... 79
  5.6 Using Pointers with Class Objects .............................. 82
  5.7 Classes with Dynamically Allocated Memory ............... 82
  5.8 Key Concepts .......................................................... 87

6 Linked Lists 93
  6.1 Background ............................................................. 93
  6.2 The Node Class ........................................................ 98
  6.3 The List Class .......................................................... 98
  6.4 Ordered Lists .......................................................... 104
  6.5 Linked Lists versus Arrays ....................................... 105
  6.6 Stacks .................................................................. 109
  6.7 Queues .................................................................. 110
  6.8 Recursion ................................................................. 112
  6.9 Recursion and Linked Lists ..................................... 117
  6.10 Sorting Linked Lists ............................................... 120
      6.10.1 Problem decomposition: a new way to think about recursion ....... 121
      6.10.2 Insertion Sort ................................................. 124
      6.10.3 Selection Sort ................................................ 124
      6.10.4 Merge-sort .................................................... 130
      6.10.5 Quicksort ....................................................... 134
  6.11 Linked-List Programming Exercises .......................... 134
      6.11.1 Purpose ......................................................... 134
      6.11.2 General Rules ................................................ 134
      6.11.3 The Exercises ............................................... 136
      6.11.4 Suggestions ................................................... 137
  6.12 Key Concepts ........................................................ 137

7 Inheritance and Polymorphism 139
  7.1 Inheritance ............................................................... 139
  7.2 The Inheritance Hierarchy ......................................... 140
  7.3 Inheritance Types/Member Access Specifiers ................ 142
  7.4 Library Hierarchy Implementation ............................... 142
  7.5 Polymorphism .......................................................... 146
  7.6 Non-virtual functions ................................................. 149
  7.7 Virtual functions ........................................................ 152
  7.8 Virtual destructors .................................................... 152
  7.9 Abstract Base Classes .............................................. 153
  7.10 Key Concepts ........................................................ 157
CONTENTS

A CSC 214 Rules 163
A.1 Academic Integrity in CSC 214 .......................... 163
A.1.1 Frequently Asked Questions about Academic Integrity in Programming Courses 163
A.1.2 The Academic Integrity Policy for CSC 214 ........................................ 164
A.2 CSC 214 Documentation and Style Standards ................................. 165
A.2.1 Documentation standards ........................................... 166
A.2.2 Style Guidelines ................................................... 169

B Dealing with Errors 173
B.1 Common Errors in C++ Programming ........................................ 173
B.1.1 General Strategies .................................................. 173
B.1.2 Actual Error Messages from the g++ Compiler ............................. 175
B.2 Using the GDB Debugger ............................................... 176

C The sXwin Window Library 179

D Useful Unix Utilities 185

E Reference Guide 189
List of Figures

1.1 Refining step 2 of the pseudo-code for the sorting program. .......................... 13
1.2 Further refinement of step 2.1. ................................................................. 13
1.3 The result of step-wise refinement on the design of the name sorting program. .... 14
1.4 Name sorting program based on top-down design and stepwise refinement (first part). 15
1.5 Name sorting program based on top-down design and stepwise refinement (continued). 16
1.6 Associating methods with objects. .............................................................. 17
1.7 Name sorting program based on object-oriented design (first part). ................ 18
1.8 Name sorting program based on object-oriented design (continued). ............. 19
1.9 Test driver for the insert function. ............................................................. 25
1.10 A bug in the sorting program (input data and output). ............................... 25
1.11 Debugging printout added to name sorting program. ................................... 27
1.12 First part of output after debugging printout is added. .............................. 28
1.13 Corrected version of the insert function (based on top-down design). .......... 29
1.14 Name sorting program that sorts “in place”. ........................................... 30
1.15 Output from the “in place” sorting program (same data as in Figure 1.10). .... 30
1.16 Debugging printout added to the “in place” sorting program. .................... 31
1.17 Debugging output to figure out where an error occurs. ............................ 32
1.18 Output from more detailed debugging printout. ....................................... 32
1.19 Correcting aliasing of strings in the insert function. ............................... 33

2.1 The file averages.cpp. .................................................................................. 36
2.2 The file sort.cpp. ......................................................................................... 37
2.3 The file sort.h. ............................................................................................ 40
2.4 Separate compilation and linking. ............................................................... 42
2.5 The makefile. .............................................................................................. 43
2.6 The makefile using GNU syntax. ............................................................... 47
2.7 Options for the g++ compiler. .................................................................... 48

3.1 A finite-state machine model of a soda dispenser. ..................................... 50
3.2 Finite-state machine showing the operation of an automated teller. .......... 50
3.3 Finite-state machine for a valid Pascal real constant. ............................... 51
3.4 Finite-state machine to search for the string “abba”. .................................. 52
3.5 Progression from state to state as a finite-state machine searches for the string “abba”. 52
3.6 Finite-state machine for a word counter. ............................................... 52
3.7 A word count program based on the finite-state machine in Figure 3.6. ...... 53
LIST OF FIGURES

3.8 Finite-state machine for printing words that are not in C++ comments. .............. 54
3.9 The program for the FSM in Figure 3.8. ................................................. 55
3.10 A simplified version of the word count program. ..................................... 56
3.11 Finite-state machine for converting a string to a number (using Horner’s rule). .... 58
3.12 Table form of the FSM for converting a string to a number. ....................... 58
3.13 Program for converting strings to numbers, based on the FSM in Figure 3.11. .... 59

4.1 Date class header file. ................................................................. 62
4.2 Date class implementation file. ...................................................... 63
4.3 Main program for dates program. .................................................... 66
4.4 Makefile for dates program. ........................................................ 67
4.5 Output from dates program. ........................................................ 67
4.6 Date List class header file. ......................................................... 68
4.7 Date List class implementation file. .............................................. 69
4.8 main function for datelist program. .............................................. 71
4.9 makefile for datelist program. .................................................... 72
4.10 Output from datelist program. ..................................................... 72

5.1 An example of a memory layout. .................................................. 76
5.2 Memory diagram for the layout presented in Figure 5.1. ......................... 76
5.3 Different interpretations for * and &. ............................................. 77
5.4 A program that causes a dangling pointer. ......................................... 78
5.5 Three different kinds of memory allocation. ....................................... 81
5.6 A dangling pointer that results from automatic memory deallocation. ............ 81
5.7 Name class header file. .............................................................. 83
5.8 Name class implementation file. ................................................... 84
5.9 Deep copy of pointer data. ........................................................... 86
5.10 Shallow copy of pointer data. ..................................................... 86
5.11 name program main function. ...................................................... 88
5.12 name program output. ............................................................... 88
5.13 Compiler errors using private copy constructor/overloaded assignment operator. 89

6.1 List class header file. ................................................................. 94
6.2 List class implementation file (first part). ....................................... 95
6.3 List class implementation file (continued). ...................................... 96
6.4 List class implementation file (continued). ...................................... 97
6.5 Ordered List class header file. ................................................... 105
6.6 Ordered List class implementation file (first part). ......................... 106
6.7 Ordered List class implementation file (continued). ....................... 107
6.8 Memory diagrams illustrating operations on an ordered linked list. .......... 108
6.9 The operations of Figure 6.8 on a dynamic array. .............................. 108
6.10 Stack class header file. ............................................................ 110
6.11 Stack class implementation file (first part). .................................. 111
6.12 Stack class implementation file (continued). .................................. 112
6.13 Queue class header file. ........................................................... 113
6.14 Queue class implementation file (first part). ................................ 114
LIST OF FIGURES

6.15 Queue class implementation file (continued) .................. 115
6.16 The recursive sort function for insertion sort ...................... 125
6.17 The recursive insert function for insertion sort ...................... 125
6.18 Trace of insertion sort on [3 2 1 4] .......................... 126
6.19 How do you get the two pieces (first and rest) of a sorted list? ........ 127
6.20 The recursive functions for selection sort .................. 128
6.21 Recursive selection sort on an array .......................... 129
6.22 How do you recombine two sorted lists? .................. 130
6.23 The recursive functions for mergesort .......................... 131
6.25 The structure of recursion for merge-sort .......................... 132
6.26 A trace of mergeSort on a larger list .......................... 133
6.27 The recursive functions for Quicksort .................. 135

7.1 The library Book class inheritance hierarchy .................. 139
7.2 Book class header file ........................................... 140
7.3 Book class implementation file .................................... 141
7.4 Reference Book class header file .......................... 142
7.5 Reference Book class implementation file .................. 143
7.6 North Carolina Reference Book class header file ............ 144
7.7 North Carolina Reference Book class implementation file .... 145
7.8 Circulating Book class header file .......................... 146
7.9 Circulating Book class implementation file .................. 147
7.10 A client program for the Book inheritance hierarchy ........ 148
7.11 Output from the program in Figure 7.10 .................... 149
7.12 Slicing when a derived class instance is copied to a base class instance .......................... 150
7.13 No slicing occurs when pointers are copied .................. 151
7.14 Book class header file with virtual print function ........ 151
7.15 Shape class header file ........................................... 153
7.16 Shape class implementation file .......................... 154
7.17 Rectangle class header file ........................................... 154
7.18 Rectangle class implementation file .......................... 155
7.19 Triangle class header file ........................................... 155
7.20 Triangle class implementation file .......................... 156
7.21 Circle class header file ........................................... 156
7.22 Circle class implementation file .......................... 157
7.23 shape program main function .......................... 158
7.24 List class header file ........................................... 158
7.25 List class implementation file .......................... 159
7.26 makefile for shape program ........................................... 160
7.27 shape program output ........................................... 160

C.1 The sxwin class header file ........................................... 180
C.2 Constants for the sxwin class ........................................... 181
C.3 sxwin member functions that respond to input events or allow the programmer to gather information about a window ........................................... 182
C.4 Drawing functions of the `sxwin` class. ........................................ 182
C.5 Functions that manipulate text in `sxwin` windows. .......................... 183
C.6 Measures of a font. ........................................................................ 183
C.7 An event loop for a program that uses `sxwin`. ................................. 184
D.1 Options for the `a2ps` utility. .......................................................... 186
E.1 ASCII Character set. ........................................................................ 190
E.2 Output from stream manipulation program. ...................................... 196
E.3 Stream manipulation using `iomanip.h`. ........................................... 197
Chapter 1

Software Design

1.1 Algorithms and Problem Solving

An algorithm is a method for solving a problem. Algorithms are independent of any programming language. They can be tested using pencil and paper without aid of a computer.

Algorithms are the basic building blocks of software. The goal of software design is to break a programming task down into problems that can be solved either by known algorithms or by ones that the programmer can easily devise. Then each algorithm can be coded in the programming language (usually as a separate function), tested, and combined with other algorithms to make up the finished software.

Consider, for example, the task of sorting a list of names alphabetically. There are many known algorithms for this problem, but let us assume they are yet to be discovered. We begin with a simple idea: the names need to be rearranged so that each name is (lexicographically) greater than the one before it and less than the one after it. A simple and rather inefficient algorithm is:

FOR every possible rearrangement $X_1, X_2, \ldots, X_k$ of the list,
IF $X_1 < X_2 < \ldots < X_k$, output $X_1, \ldots, X_k$

This algorithm is described using pseudo-code, a notation that borrows control structures such as if-then-else, while, and for from programming languages and expresses data manipulation in English or standard mathematics.

An algorithm leaves many details unspecified. Sometimes these are minor and do not stand in the way of pencil-and-paper testing. For example, without knowing how to check whether one name is (lexicographically) less than another in C++, we can easily do so by just looking at the names. Thus, the IF $X_1 < X_2 < \ldots < X_k$ test does not require further elaboration at this time.

But how do we generate all possible rearrangements of $k$ names? We need to know this in order to test our algorithm. \footnote{One might argue that the algorithm is so obviously correct that it needs no testing, but until a way is found to generate the rearrangements, the algorithm is really of no use.} And this is not an easy task (although it is possible).

Another algorithm for the same problem is the following:
initialize the list to contain $X_1$
FOR $i = 2$ to $k$,
assume $X_1, \ldots, X_{i-1}$ are already sorted
and insert $X_i$ into the correct position among them

This algorithm (called insertion sort) is ready for pencil-and-paper testing, especially if we write
each name on a card.

Unlike programs, algorithms need not adhere to any rigid syntax. For example, the insertion
sort algorithm could just as easily have been expressed as:

\begin{verbatim}
let L be a list that has only one of the names in it
take each other name in turn and insert it into its proper position
among the names already in L
\end{verbatim}

Clarity, simplicity, and testability are the main criteria. Furthermore, the level of detail in an
algorithm can range from very abstract (as illustrated by the above examples) to “almost a program”.
The next section addresses the transition from high-level algorithms to programs.

### 1.2 Top-Down Design and Stepwise Refinement

Two important details that have been omitted in the algorithms of the previous section are the
handling of input and output. We assume for simplicity that names are delimited (separated from
each other) by white space (this works if we deal with, say, last names only) and that output should
be one name per line. Furthermore, we use standard input and output.

At the top level we can describe the algorithm for the overall program as:

1. read all names from standard input
2. sort the names lexicographically using insertion sort
3. write the names, one per line, in sorted order

The pattern is typical of many programs — read (and store) input, process data, write output. The
next part of the design process is to spell out each step in more detail, continuing to do so until the
design is broken down into steps that we know how to program directly. The approach we are using
is called top-down design with stepwise refinement.

The more experienced we are, the less we need to spell out in the design process. For most
seasoned programmers, the above three steps, along with the input assumptions, are sufficient to
enable a program to be written. Suppose we have almost no experience programming. The first
decision we have to make is how to store the data. That it does need to be stored should be clear
—if the names appear in reverse order, for example, we cannot output anything until the last input
name has been read. The only option we know about at this point is an array of C-strings, a two-
dimensional array of characters. Let $\text{MAX}_NAMES$ be the maximum number of names we want to deal
with and let $\text{name\_array}$ be an array of (up to) that many names. When there are fewer names, we
will want to know how many there are, so let $\text{number\_of\_names}$ be the actual number from input.

Stepwise refinement means breaking down each step of an algorithm into smaller, more de-
tailed, steps. Having decided to use an array for data storage, we can now elaborate on steps 1 and
3 of the previous algorithm.
2.0. let new_name_array be the sorted copy of the names
   and new_name_array[ 0 ] = name_array[ 0 ]
2.1. for index = 1 to number_of_names - 1,
   insert name_array[ index ] into its proper position among
   new_name_array[ 0 ], ..., new_name_array[ index - 1 ]
2.2. copy new_name_array back to name_array (or use new_name_array in step 3)

Figure 1.1: Refining step 2 of the pseudo-code for the sorting program.

2.1.1. let target_index = position where name_array[ index ] belongs among
       new_name_array[ 0 ], ..., new_name_array[ index - 1 ]
2.1.2. shift new_name_array[ target_index ], ..., new_name_array[ index - 1 ]
       one position to the right
2.1.3. copy name_array[ index ] to new_name_array[ target_index ]

Figure 1.2: Further refinement of step 2.1.

1.0. index = 0
1.1. while there is still input
       read next name into name_array[ index ]
       increment index
1.2. number_of_names = index

and

3.1. for index = 0 to number_of_names - 1,
       write name_array[ index ]
       write end_of_line

For step 2 we first plug in the insertion sort algorithm as shown in Figure 1.1. We can express the loop body under step 2.1 in even more detail. Figure 1.2 shows how. Putting it all together leaves us with the result shown in Figure 1.3.

At this point we translate the various steps into C++ as illustrated by the program in Figures 1.4 and 1.5. Some steps are implemented as separate functions while others show up as one or more statements within a function. Function names and comments are used to relate program parts to steps of the design. Knowledge of the language and careful attention to detail are required here. For example, it is important to know that `strcmp` and `strcpy` must be used in place of relational operators and assignment to compare and copy strings. Or, without these string library functions, we would be forced to write our own equivalents.

The program in Figures 1.4 and 1.5 has a bug. Can you find it? We will take a closer look in Sections 1.5 and 1.6.

### 1.3 Object-Oriented Design

Design methodologies are tools to guide our thinking about a programming task. **Object-oriented design** is an additional tool to add to our arsenal. Contrary to the hype that often accompanies it,
1. read all names from standard input
   1.0. index = 0
   1.1. while there is still input
       read next name into name_array[index]
       increment index
   1.2. number_of_names = index
2. sort the names lexicographically using insertion sort
   2.0. let new_name_array be the sorted copy of the names
       and new_name_array[0] = name_array[0]
   2.1. for index = 1 to number_of_names − 1,
       insert name_array[index] into its proper position among
       new_name_array[0],..., new_name_array[index − 1]
       2.1.1. let target_index = position where name_array[index]
              belongs among new_name_array[0],..., new_name_array[index − 1]
       2.1.2. shift new_name_array[target_index],..., new_name_array[index − 1]
              one position to the right
   2.2. copy new_name_array back to name_array
3. write the names, one per line, in sorted order
   3.1. for index = 0 to number_of_names,
       write name_array[index]
       write end_of_line

Figure 1.3: The result of step-wise refinement on the design of the name sorting program.

object-oriented design should not replace other design methods. Nor is O-O design limited to use
with object-oriented languages such as C++ and Java. Ideally, design and implementation should
be independent of each other.

To illustrate the differences between top-down and object-oriented design we now examine the
name sorting program from an object-oriented point of view. The first step is to identify the most
prominent nouns and verbs that occur in the problem description (or in descriptions of algorithms
to be used). The key nouns appear to be “name” and “list” (of names), while the key verbs are
“read”, “write”, and “sort”. If we decide on insertion sort, the verb “insert” should also be included.
The nouns are called objects. When there is more than one occurrence, or instance, as is the case
with “name”, the noun identifies a class2 of objects. The verbs are called methods, actions that
are applied to the objects (or that the objects are capable of).

The second step is to associate methods with objects. Figure 1.6 illustrates the end result of
this process. Some arbitrary decisions were made — recall that the methodology is a thinking tool:
will not tell you how to write the program. A name is implemented as a C-string and a list is
implemented as an array of C-strings. This means that all of the name methods can be implemented
directly in terms of known C++ constructs. A list (of names) is an array of names along with an
integer specifying how many names there are. The integer is needed because, unlike a C-string, the
array does not have a sentinel value (for example, the null, or \0 that marks the end of a string).
The insert operation involves both a name and a list, but only the list is changed by it; hence the
decision to include it only as a method for the list object.

In thinking about the list object, we decided that it should be kept in sorted order from the
beginning. Instead of reading the list and then sorting it, we decided to read each name and insert it

2It will not necessarily be implemented as a C++ class, as we shall see.
const int MAX_NAMES = 512;
const int NAME_LENGTH = 511;  // standard line limit for Unix files

#include <iostream.h>
#include <string.h>

int read_names( char name[][ NAME_LENGTH + 1 ] );
// returns the number of names read

void sort_names( char name[][ NAME_LENGTH + 1 ], int length );

void write_names( char name[][ NAME_LENGTH + 1 ], int length );

int main()
{
    char name_array[ MAX_NAMES ][ NAME_LENGTH + 1 ];
    int number_of_names = read_names( name_array );
    sort_names( name_array, number_of_names );
    write_names( name_array, number_of_names );
}

Figure 1.4: Name sorting program based on top-down design and stepwise refinement (first part).
int read_names( char name[][ NAME_LENGTH + 1 ] )
{
    int index = 0;
    // while there is still input, read next name into name_array[ index ]
    // and increment index; check to make sure index does not go out of
    // bounds!
    while ( index < MAX_NAMES && cin >> name[ index ] ) {
        ++index;
    }
    if ( index >= MAX_NAMES ) {
        cerr << "Warning: input may have been truncated."
             << endl;
        cerr << "Final name read was '" " << name[ index - 1 ] << "'
             << endl;
    }
    return index;
}

void insert( const char to_be_inserted[], char name[][ NAME_LENGTH + 1 ], int length )
{
    // let target_index = position where to_be_inserted belongs among
    //      name[ 0 ], ..., name[ length - 1 ]
    int target_index = length - 1;
    while ( target_index >= 0 && strcmp( name[ target_index ], to_be_inserted ) > 0 ) {
        --target_index;
    }
    // shift new_name_array[ target_index ], ..., new_name_array[ length - 1 ]
    // one position to the right
    for ( int i = length - 1; i >= target_index; --i ) {
        strcpy( name[ i + 1 ], name[ i ] );
    }
    // copy to_be_inserted to name[ target_index ]
    strcpy( name[ target_index ], to_be_inserted );
}

void sort_names( char name[][ NAME_LENGTH + 1 ], int number_of_names )
{
    char new_name[ MAX_NAMES ][ NAME_LENGTH + 1 ];
    strcpy( new_name[ 0 ], name[ 0 ] );
    for ( int i = 1; i < number_of_names; ++i ) {
        insert( name[ i ], new_name, i );
    }
    // copy new_name array back to the original name array
    for ( int i = 0; i < number_of_names; ++i ) {
        strcpy( name[ i ], new_name[ i ] );
    }
}

void write_names( char name[][ NAME_LENGTH + 1 ], int number_of_names )
{
    for ( int i = 0; i < number_of_names; ++i ) {
        cout << name[ i ] << endl;
    }
}

Figure 1.5: Name sorting program based on top-down design and stepwise refinement (continued).
### 1.4 THE “WATERFALL” MODEL

<table>
<thead>
<tr>
<th>method</th>
<th>description</th>
<th>implementation</th>
</tr>
</thead>
<tbody>
<tr>
<td>read</td>
<td>to input each name</td>
<td>&gt;&gt; operator</td>
</tr>
<tr>
<td>write</td>
<td>to write each name</td>
<td>&lt;&lt; operator</td>
</tr>
<tr>
<td>copy</td>
<td>copies name from one place to another</td>
<td><code>strcpy</code></td>
</tr>
<tr>
<td>compare</td>
<td>to decide whether one name lexicographically precedes another</td>
<td><code>strcmp</code></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>List</th>
<th>read</th>
<th>read and store all names</th>
<th>not used</th>
</tr>
</thead>
<tbody>
<tr>
<td>write</td>
<td>write all names after sorting</td>
<td>separate function</td>
<td></td>
</tr>
<tr>
<td>sort</td>
<td>sort the list of names</td>
<td>not used</td>
<td></td>
</tr>
<tr>
<td>insert</td>
<td>insert a new name into the list</td>
<td>separate function</td>
<td></td>
</tr>
</tbody>
</table>

Figure 1.6: Associating methods with objects.

into the list in the proper position. This means that only two list methods need to be implemented, one for inserting and one for writing.

A program based on this design is in Figures 1.7 and 1.8. The details are very similar to the program based on top-down design. Overall organization of the code, comments, and naming conventions are used to emphasize the object-oriented design. For example, the list data is global so it can be accessed by all the list methods and the list methods have names starting with `List_`. This program has the same bug as the one based on top-down design.

1.4 The “Waterfall” Model

The **waterfall model** of software production is a list of stages in the development of a program. Each stage makes subsequent stages possible but no stage is ever completed. Since each stage flows into the next but also overlaps it in time, a time-sequence diagram of the stages often looks like a series of cascades, hence the name “waterfall model”.

Software engineering texts differ in their naming of the stages, but what follows is a typical list:

**Requirements.** The circumstances necessitating the software are analyzed. Is this program really needed? What existing software or manual procedure does it replace? What criteria does it need to satisfy?

**Specification.** A detailed description of the behavior of the software from the user point of view is developed. What are the exact inputs and outputs? How should it look and feel under normal circumstances? How should it handle mistakes on the part of the user?

**System Design.** The software is divided into smaller, more manageable, parts and the functions of each part are clearly defined. In top-down design, the first subdivision is typically into input, processing, and output. In object-oriented design, the parts correspond to objects and classes.

**Detailed Design.** This stage bridges the gap between system design and coding by outlining how the system design will be rendered in the chosen programming language (all previous stages

---

3We could have had the list and number of names as arguments to all methods, or, as we will learn later, declared a list class.
//: sort_names.cpp - program to sort a list of names (object-oriented
//:                  design version)
// Matt Stallmann, 17 Jun 1999

// INPUT (standard input): a list of no more than MAX_NAMES names,
// separated by white space (blanks, tabs, newlines). Each name has no more
// than NAME_LENGTH characters.

// OUTPUT (standard output): the input names listed in lexicographic order,
// one per line.

// COMPILe: g++ -g -Wall sort_names.cpp -o sort_names
// USAGE: sort_names < input_file > output_file (or use standard I/O)

const int MAX_NAMES = 512;
const int NAME_LENGTH = 511;    // standard line limit for Unix files

#include <iostream.h>
#include <string.h>

// methods for the list object
void List_insert( const char name[] );
void List_write();

int main()
{
    char name[ NAME_LENGTH + 1 ];
    while ( cin >> name ) {
        cin >> name ;
        List_insert( name );
    }
    List_write();
}
1.4. THE “WATERFALL” MODEL

// Implementation of the List object:
// 1. Data for the list object (global so it can be accessed by all methods)
char name_array[ MAX_NAMES ][ NAME_LENGTH + 1 ];
int number_of_names = 0;

// 2. Methods for the List object
void List_insert( const char name[] )
{
    // issue warning and do nothing if this would lead to overflow
    if ( number_of_names >= MAX_NAMES ) {
        cerr << "Warning: too many names − " << name << " was not inserted."
        << endl;
        return;
    }

    // let target_index = position where name belongs among
    // name_array[ 0 ], ..., name_array[ number_of_names − 1 ]
    int target_index = number_of_names − 1;
    while ( target_index >= 0 && strcmp( name_array[ target_index ], name ) > 0 ) {
        --target_index;
    }

    // shift name_array[ target_index ], ..., name_array[ number_of_names − 1 ]
    // one position to the right
    for ( int i = number_of_names − 1; i >= target_index; −−i ) {
        strcpy( name_array[ i + 1 ], name_array[ i ] );
    }

    // copy name to name_array[ target_index ]
    strcpy( name_array[ target_index ], name );

    // increment number of names
    ++number_of_names;
}

void List_write( )
{
    for ( int i = 0; i < number_of_names; ++i ) {
        cout << name_array[ i ] << endl;
    }
}

Figure 1.8: Name sorting program based on object-oriented design (continued).
are language-independent). When the language is C++, this step usually involves the writing of function headers or **prototypes**.

**Coding (Implementation).** The actual writing of the various parts of the program — in a typical software project less than 20% of the total time is spent in this stage. Coding should be tightly integrated with the other stages. Like verification (the next stage), it begins on the first day of a project and continues as long as the software is in use.

**Verification.** Also referred to as **validation**\(^4\), this stage ensures that software meets its requirements. Verification should occur during all of the other stages and will be addressed in more detail in the next section.

Another aspect of software production that permeates all stages is **documentation**. The results of the reasoning that goes on in every stage must be written down and eventually reflected in code and comments. Large software projects often have separate requirements documents spanning hundreds of pages and other documents that describe every step of the design. For the CSC 210 programming assignments it is sufficient to have a **README** file that explains specifications and system design\(^5\) and comments within programs that reflect detailed design. See Appendix A for more details.

We now take a look at the sorting example from the previous two sections, examining each stage of the waterfall model in detail.

**Requirements.** The requirements stage might establish the need to sort names occurring in various documents alphabetically. An important question to ask at this point is whether an already existing program will solve the problem. There is, for example, a Unix **sort** utility. The only drawback of the **sort** utility is that it requires one name per line in the input, while the documents whose names we want to sort might have several names on each line (do **man sort** for more details). So all we really need is a simple program to put one name on each line (this can also be done with Unix utilities, but is actually simpler in C++);

```cpp
#include<iostream.h>
int main()
{
    char word[ 512 ];
    while ( cin >> word ) cout << word << endl;
}
```

After compiling this program and calling it **list_words**, we can accomplish the original task by running **list_words | sort**, or if we have an input file **in.dat** and want output to go to **out.dat**, the command would be

```
unity% list_words < in.dat | sort > out.dat
```

Problem solved.\(^6\) See Appendix D for an explanation of input/output redirection.

**Specification.** In the specification stage we need to clarify the formats of the input and output and the definition of lexicographic order. Among the questions we need to answer are:

---

4. “Validation” usually suggests an engineering approach based on testing while “verification” suggests logical analysis. Sometimes this stage is referred to as “V&V” so as to include both.

5. The first few assignments will not require a **README** file. Specifications should be included in the **README** file only if they go beyond what is stated in an assignment.

6. In CSC 210 programming assignments, the standard assumption is that all Unix utilities are broken, as is other off-the-shelf software. Otherwise our requirements might lead to solutions that involve no C++ programming!
1.5 Verification: Logical and Experimental

- What constitutes white space?
- How many characters are in the longest possible name and what should happen if this bound is exceeded?
- How many names can there be and what should happen if there are more?
- Can a name contain characters other than alphabetic ones? If so, how should they be treated when sorting?

The programs presented earlier take the easy way out on all of these questions: white space is defined according to C++ input conventions (blanks, tabs, or newlines); behavior becomes unpredictable if a name is too long (the array bound is exceeded without warning); the number of names is easy to check, so we print a warning and stop reading when there are too many; and lexicographic order is defined by strcmp so that any non-alphabetic characters are sorted using the ASCII collating sequence (see Appendix E). The Unix sort utility has options to allow other orderings (e.g. ignore the case of letters, ignore non-alphanumeric characters).

System Design, Detailed Design, and Coding. We have seen examples of system design, detailed design, and coding in the previous two sections. Even the simplest programming task has many equally good solutions. The techniques presented in this chapter are tools to guide your thinking about programs and to help you build larger programs from smaller ones. No technique, however, can supply the creativity and careful attention to detail required to solve a programming problem. The next section examines implementation, testing, and debugging strategies that lead to a more manageable programming task.

1.5 Verification: Logical and Experimental

There are two ways to reassure yourself of the correctness of a program: one is to convince yourself by logic that every possible input has been handled correctly; the other is to test the program with enough sample data. Neither of these is adequate by itself. The first fails when you make careless errors in logic or have misunderstandings about how C++ works, the second fails because it’s hard to know how much or what particular sample data is enough.

In addition to needing both logical and experimental verification, you need to be able to compartmentalize verification to small parts of a large program rather than dealing with the complexity of the whole. Furthermore, verification should take place at every stage of software development: specifications can be subjected to scenarios in brainstorming sessions, system design can be checked for coverage of all possibilities by doing a walk through (use each scenario from the specification stage to determine precisely what role each system component will play), and so on. The rest of this section emphasizes ways to incorporate verification into implementation.

When you write a large program it is important to write one small piece at a time. You already figured out what the pieces should be when you did a system design. Now you can develop strategies that allow you to verify each piece soon after it is written instead of waiting until the whole program is put together. There is an added psychological bonus: the satisfaction of getting something working correctly after only a small amount of effort.

**Contract Programming.** You can separate the logic of different program parts more easily if you think of each function as a contract between the **client** (part of program that calls the function)
and implementation. For example, consider the `insert` function in the top-down design version of the name sorting program. It should have been commented, possibly as follows (see [PL96]):

```c
void insert( const char to_be_inserted[], char name[] [ NAME_LENGTH + 1 ],
            int length )
// Arguments:
// to_be_inserted - the C-string to be inserted (duh)
// name - an array of C-strings
// length - number of names in the name array
// Side effects:
// to_be_inserted is added to the name array in the appropriate
// lexicographic position;
// items of the name array that follow to_be_inserted in lexicographic order
// are shifted one position to the right
```

Seeing this as a contract allows you to assume that `insert` does the right thing and you can verify the `sort` function without looking at the details of `insert`. Everything is OK if the implementation of `insert` holds up its end of the bargain.

The contract has two sides, however. In this case you can’t expect the implementation to do the right thing without knowing the circumstances under which `insert` is called. It is not possible to shift array items to the right unless you know the array has enough room for them. The “appropriate lexicographic position” makes no sense if the items in `name` are not already sorted. Finally, `to_be_inserted` needs to be short enough to fit into the array. The client’s obligations are usually put in the form of assumptions:

```c
// Assume: 1. The items in the name array are sorted lexicographically;
// 2. The name array has room for at least length+1 entries
// 3. strlen(to_be_inserted) <= NAME_LENGTH
```

or the first assumption could be folded into the description of the arguments:

```c
// Arguments:
// to_be_inserted - the C-string to be inserted (duh)
// name - an array of C-strings, sorted lexicographically
// length - number of names in the name array
```

Now the contract is complete: the client promises to deliver an array of lexicographically sorted C-strings with at least one extra slot and a string to be inserted that is not too long; the implementation promises to `insert to_be_inserted` into the proper position, shifting lexicographically larger items to the right. The contract allows both the logical verification and the testing of the client and implementation to be separated from each other (we’ll say more about the testing later).

A more precise way to describe the contract is via **preconditions** and **post-conditions**. A precondition is a logical description of what should be true **before** a function begins execution, that is, the promise made by the client. A post-condition describes what is true **after** the function is executed, the implementation’s promise. See Headington and Riley [HR94] for more details. A simple rewriting of the contract for `insert` using pre- and post-conditions is (there are many variations in the notation; we arbitrarily chose that of Headington and Riley):

```c
1.5. **VERIFICATION: LOGICAL AND EXPERIMENTAL**

```c
void insert( const char to_be_inserted[], char name[][ NAME_LENGTH + 1 ],
            int length )
// PRE: strlen(to_be_inserted) <= NAME_LENGTH
//     && to_be_inserted has at least length + 1 rows
//     && items 0,...,length-1 of name are lexicographically sorted
// POST: name is a lexicographically sorted array of length + 1 items, the
//     && name contains the 'length' items of name<PRE> and to_be_inserted
```

The wording is changed to emphasize the fact that both pre- and postconditions should describe the data rather than the process that is applied to it (“the name array is sorted and it contains ...” instead of “to_be_inserted has been added and ... have been shifted to the right”). The usage name<PRE> refers to name as it was at the time of the precondition. A contract is usually simpler and easier to enforce if it describes what the end product should look like instead of how it should get to be that way.

Now we’re ready to address implementation strategies that allow separate testing of program parts. In general testing aims for adequate **test coverage**. A good rule of thumb is that you try to come up with data that forces every alternative of an if or switch to be executed and every loop to be executed 0 times (if possible), once, and twice. Then you can do **industrial-strength testing**: generate lots of randomly-generated data to make it likely that every variation has been presented to the program. Both ideas can be applied to the testing of program parts and the testing of the whole program when it has been completed.

Regardless of your design methodology, implementation and testing can be done either top-down or bottom-up (or a combination of the two) to isolate smaller program parts.

**Top-Down Implementation and “White Box” Testing.** To implement a program top-down you first implement the main program and write stubs for the functions it calls. A stub is an implementation of a function that does nothing other than print a message letting the user know it has been called and what the arguments are. If the function returns a value, the stub has to make up something that will allow testing with at least some data (or it can ask the user to input the desired value). A stub for the sort_names function in the top-down design version of the name sorting program might look like:

```c
void sort_names( char name[][ NAME_LENGTH + 1 ], int number_of_names )
{
    cout << "Entering sort_names, number_of_names = " << number_of_names << endl;
    cout << " name array = " << endl;
    for ( int i = 0; i < number_of_names; ++i )
        cout << " " << name[ i ] << endl;
}
```

It makes no sense to defer full implementation of read_names and write_names functions: you need them to be able to do any sensible testing. With a stub for sort_names and fully written read_names and write_names you have a program that reads data and writes it in the same order. You can feel accomplished when you get this working. You have gotten past many potential syntax errors. Not only do you know that input and output are working correctly, but you have a scaffolding for finishing the implementation of sort_names, which you know is being called correctly. Don’t throw away the output statements in sort_names — they can be useful for later testing and debugging (see Section 1.6).

Next you implement the sort_names function and write the following stub for insert:
void insert( const char to_be_inserted[], char name[][ NAME_LENGTH + 1 ],
int length )
{
    cout << "Entering insert, to_be_inserted = " << to_be_inserted << endl;
    cout << " length = " << length << endl;
    cout << " name array = " << endl;
    for ( int i = 0; i < length; ++i ) cout << " " << name[ i ] << endl;
}

This allows you to make sure that insert is being called as many times as it needs to be and with
the proper arguments each time. We call this **white box testing** because each function in turn is
treated as an empty (white) box and we test to make sure it is being called correctly by the client.
In a large program stubs are repeatedly replaced by full or partial implementations (in a complex
function you may not want to deal with all cases right away) and stubs are written for functions
that are called but have not been implemented.

**Bottom-Up Implementation and “Black Box” Testing.** Bottom-up implementation starts
with functions that don’t call any other (user-defined) functions, read_names, write_names, and
insert in the case of name sorting based on top-down design. To test each function you treat it as a
“black box”: there is stuff (functionality) inside but you can’t see it. So you write a **test driver**, a
main program that allows you to feed arguments to the function and see the results.

For top-down design name sorting the main program without the call to the sort_names function
serves as a good test driver for both read_names and write_names (no sense in testing these
separately since testing each would require the equivalent of the other). A test driver for insert
might work as is illustrated in Figure 1.9. It can be run many times with different inputs before
it is used with sort_names, thus allowing you to detect bugs in insert before putting the whole
program together. The advantage of this can be seen in the next section, where part of the challenge
of debugging is figuring out which function is the culprit. As programs get large and complex, the
extra effort of writing test drivers pays off.

### 1.6 Debugging

We now look at two case studies in debugging to illustrate both logical and experimental approaches.
The bugs illustrated here were actually encountered during implementations of the name sorting
program.

**Case Study 1: Be careful with array indices.** The program in Figures 1.4 and 1.5 (based on
top-down design) has a subtle bug. You compile the program:

```
unity% g++ -Wall -g sort_tds.cpp -o sort_tds
```

and run it redirecting input from the data file sort.dat. Figure 1.10 shows the contents of sort.dat
and the output from the program.

Something is wrong here: Frank doesn’t appear in the output, there’s a blank line, and the output
isn’t even close to being sorted. To narrow the search for bugs, you add **debugging printout** to
each of the functions — see Figure 1.11. Only the code for the informative printout at the beginning
and end of each function is shown. The #ifdef DEBUG ... #endif brackets code that will be
compiled only if the compile-time constant DEBUG is defined. You can compile the program like
int main()
{
    // get data for the arguments
    cout << "How many names?" << endl;
    int length;
    cin >> length;
    cout << "What are they (lexicographic order, please)?" << endl;
    char name_array[MAX_NAMES][NAME_LENGTH + 1];
    for (int i = 0; i < length; ++i) cin >> name_array[i];
    char to_be_inserted[NAME_LENGTH + 1];
    cin >> to_be_inserted;

    // call the function
    insert(to_be_inserted, name_array, length);

    // output the result (side effect)
    cout << "After insertion, the array is:" << endl;
    for (int i = 0; i <= length; ++i)
        cout << name_array[i] << endl;
}

Figure 1.9: Test driver for the insert function.

unity% cat sort.dat
Joey Frank
Mabel  Bert Theresa
Joan
Allison

unity% sort_tds < sort.dat
Bert
Joan
Allison

Mabel
Theresa
Joey
unity%

Figure 1.10: A bug in the sorting program (input data and output).
unity% g++ -Wall -g -DDEBUG sort_tds.cpp -o sort_tds

to include the messages (-Dx has the effect of defining the constant x), or leave out the -DDEBUG to omit them. To be most useful, the messages should tell you when the program enters and leaves the function, what the values of the arguments are at the time of entry, and what the values of any variables that might have changed are at the time of exit (also report the value returned, if any). Sending output to cerr instead of cout ensures that the output will be printed immediately when that part of the code is executed (instead of being stored in a buffer).

Figure 1.12 shows the first part of what the output now looks like. Several screens full of output follow, but the most prudent course is to focus on the first sign of trouble (use the scroll bar to view the output or save it in a file by redirecting both standard output and standard error — use >& instead of > in the C-shell). There is a blank line in the array where Frank should have been inserted.

Time to get out the pencil and paper. On entering insert, the string to be inserted is Frank, the name array contains Joey at index 0 and its length is 1. You draw a box to keep track of the value of target_index and write a 0 (length - 1) in it. Looking at the condition of the while loop, both parts appear to be satisfied (target_index is at least 0 and Joey is after Frank alphabetically. So the body of the loop is executed; you cross out the 0 in the target_index box and write a -1 next to it. Now the condition fails and you move beyond the loop.

*Hold on a minute!* The comment at the top of this code segment claims that target_index should be the index where the new name is to be inserted, and the rest of the function is written based on this claim. The correct index is 0 rather than -1. Aha! By the time we exit the loop, the index is actually 1 less than the correct index: the loop is terminated either when the index goes negative or when the name stored at the index should be before (or is the same as) the name we’re inserting. The “off by one” error is easily corrected. The correct version of the insert function is in Figure 1.13.

You found the bug using a combination of carefully placed output statements and analysis with pencil and paper. Had you done the pencil-and-paper work when writing the insert function, you might have discovered the bug sooner.

**Case Study 2: Watch out for aliases.** Consider a slightly different version of the name sorting program. The implementations of sort and insert are shown in Figure 1.14. Instead of inserting the names into a new array, this version inserts them into the existing array — at the beginning of every iteration of the for loop in sort_names the leftmost i items are sorted.

You are caught off guard by the output — see Figure 1.15. Next step: add debugging output as in the previous case study (perhaps you should have done so already). The first part of the output is illustrated in Figure 1.16 (again, we stop at the first sign of a problem). This time Frank appears to have changed his name to Joey during execution of insert. Puzzling. You divide and conquer: does the change occur during execution of the loop or after? Additional debugging output in the loop body and after the loop should clarify this — see Figure 1.17.

The output, starting at the first execution of insert, is shown in Figure 1.18. Now you see that the change occurs between the beginning of the first loop iteration and the first statement after the loop. It must therefore be the result of doing the

```c
strcpy( name[ position + 1 ], name[ position ] );
```

with position being 0. So Joey is copied from position 0 to position 1, the position originally occupied by Frank. But isn’t this pass by value, you think to yourself — to be inserted should be
1.6. DEBUGGING

```c++
int read_names( char name[][ NAME_LENGTH + 1 ] )
{
    #ifdef DEBUG
    cerr << "-> read_names" << endl;
    #endif
    INSERT CODE HERE...
    #ifdef DEBUG
    cerr << "[end of read] index = " << index << endl;
    cerr << " name array = " << endl;
    for ( int i = 0; i < index; ++i ) cerr << " " << name[ i ] << endl;
    cerr << "<- read_names" << endl;
    #endif
    return index;
}

void insert( const char to_be_inserted[], char name[][ NAME_LENGTH + 1 ], int length )
{
    #ifdef DEBUG
    cerr << "-> insert, to_be_inserted = " << to_be_inserted << endl;
    cerr << " length = " << length << endl;
    cerr << " name array = " << endl;
    for ( int i = 0; i < length; ++i ) cerr << " " << name[ i ] << endl;
    #endif
    INSERT CODE HERE...
    #ifdef DEBUG
    cerr << "[end of insert] name array = " << endl;
    for ( int i = 0; i <= length; ++i ) cerr << " " << name[ i ] << endl;
    cerr << "<- insert" << endl;
    #endif
}

void sort_names( char name[][ NAME_LENGTH + 1 ], int number_of_names )
{
    #ifdef DEBUG
    cerr << "-> sort_names, number_of_names = " << number_of_names << endl;
    cerr << " name array = " << endl;
    for ( int i = 0; i < number_of_names; ++i ) cerr << " " << name[ i ] << endl;
    #endif
    INSERT CODE HERE...
    #ifdef DEBUG
    cerr << "[end of sort] name array = " << endl;
    for ( int i = 0; i < number_of_names; ++i )
    cerr << " " << name[ i ] << endl;
    cerr << "<- sort" << endl;
    #endif
}
```

Figure 1.11: Debugging printout added to name sorting program.
-> read_names
[end of read] index = 7
name array =
  Joey
  Frank
  Mabel
  Bert
  Theresa
  Joan
  Allison
<- read_names
-> sort_names, number_of_names = 7
name array =
  Joey
  Frank
  Mabel
  Bert
  Theresa
  Joan
  Allison
-> insert, to_be_inserted = Frank
length = 1
name array =
  Joey
[end of insert] name array =
  Joey
<- insert
...
void insert( const char to_be_inserted[], char name[][ NAME_LENGTH + 1 ], int length )
{
    #ifdef DEBUG
        cerr << "-> insert, to_be_inserted = " << to_be_inserted << endl;
        cerr << " length = " << length << endl;
        cerr << " name array = " << endl;
        for ( int i = 0; i < length; ++i ) cerr << " " << name[ i ] << endl;
    #endif
    // let target_index = position where to_be_inserted belongs among
    // name[ 0 ], ..., name[ length − 1 ]
    int target_index = length − 1;
    while ( target_index >= 0 && strcmp( name[ target_index ], to_be_inserted ) > 0 )
    {
        --target_index;
        // at this point, target_index is one position to the left of where it
        // should be (BUG FIXED: 21 Jun 1999)
        ++target_index;

        // shift new_name_array[ target_index ], ..., new_name_array[ length − 1 ]
        // one position to the right
        for ( int i = length − 1; i >= target_index; --i )
        {
            strcpy( name[ i + 1 ], name[ i ] );
        }

        // copy to_be_inserted to name[ target_index ]
        strcpy( name[ target_index ], to_be_inserted );
    #ifdef DEBUG
        cerr << "[end of insert] name array = " << endl;
        for ( int i = 0; i <= length; ++i ) cerr << " " << name[ i ] << endl;
        cerr << "<-insert " << endl;
    #endif
    }
}

Figure 1.13: Corrected version of the insert function (based on top-down design).
void insert( const char to_be_inserted[], char name[][ NAME_LENGTH + 1 ], int length )
{
    int position = length - 1;
    while ( position >= 0 && strcmp( name[ position ], to_be_inserted ) > 0 ) {
        strcpy( name[ position + 1 ], name[ position ] );
        --position;
    }
    strcpy( name[ position + 1 ], to_be_inserted );
}

void sort_names( char name[][ NAME_LENGTH + 1 ], int length )
{
    // keep the first name in name[ 0 ] (initially)
    for ( int i = 1; i < length; ++i ) {
        insert( name[ i ], name, i );
    }
}

Figure 1.14: Name sorting program that sorts “in place”.

unity% g++ -g -Wall sort_names.cpp -o sort_names
unity% sort_names < sort.dat
Joey
Joey
Mabel
Mabel
Theresa
Theresa
Theresa
unity%

Figure 1.15: Output from the “in place” sorting program (same data as in Figure 1.10).
1.7. KEY CONCEPTS

a copy of the string Frank, it has nothing to do with the original name array. You’d be right if these were well-behaved strings (like the built-in string class now available on many C++ compilers). But these are C-strings, which, like any arrays in C, are passed by reference. The insert function is given the address of the to_be_inserted argument, in this case the address of position 1 in the name array. In pass by reference the dummy argument (formal parameter), in this case to_be_inserted, is just an alias, or another name, for the actual argument (parameter), in this case name[i]. By the time the actual insertion takes place, the data at that address has been changed by the strcmp.

A simple solution is to make an explicit copy of to_be_inserted, as illustrated in Figure 1.19.

1.7 Key Concepts


Contract programming - Type of programming in which the client of a function agrees to meet certain preconditions and the implementor of a function agrees to meet certain postconditions. The preconditions state what must be true in order for the function to work correctly and the postconditions state what will be true after the function successfully completes execution.

Top-down design with step-wise refinement - Method of designing an algorithm by initially describing it as a sequence of high-level steps. A step is successively refined (itself described

Figure 1.16: Debugging printout added to the “in place” sorting program.
```c
void insert( const char to_be_inserted[], char name[][ NAME_LENGTH + 1 ], int length ) {
    #ifdef DEBUG
    cerr << "-> insert, to_be_inserted = " << to_be_inserted << endl;
    cerr << "length = " << length << endl;
    cerr << "name array = " << endl;
    for ( int i = 0; i < length; ++i ) cerr << " " << name[ i ] << endl;
    #endif
    int position = length - 1;
    while ( position >= 0 && strcmp( name[ position ], to_be_inserted ) > 0 ) {
        #ifdef DEBUG
        cerr << "loop: position = " << position
        << ", name[ " << position << "] = " << name[ position ]
        << ", to_be_inserted = " << to_be_inserted << endl;
        #endif
        strcpy( name[ position + 1 ], name[ position ] );
        --position;
    }
    #ifdef DEBUG
    cerr << "out of loop: position = " << position
    << ", name[ " << position + 1 << "] = " << name[ position + 1 ]
    << ", to_be_inserted = " << to_be_inserted << endl;
    #endif
    strcpy( name[ position + 1 ], to_be_inserted );
    #ifdef DEBUG
    cerr << "[end of insert] name array = " << endl;
    for ( int i = 0; i <= length; ++i ) cerr << " " << name[ i ] << endl;
    cerr << "<- insert" << endl;
    #endif
}
```

Figure 1.17: Debugging output to figure out where an error occurs.

```
-> insert, to_be_inserted = Frank
length = 1
name array = 
Joey
loop: position = 0, name[ 0 ] = Joey
to_be_inserted = Frank
out of loop: position = -1, name[ 0 ] = Joey, to_be_inserted = Joey
[end of insert] name array = 
Joey
Joey
<- insert
```

Figure 1.18: Output from more detailed debugging printout.
void insert( const char to_be_inserted[], char name[][ NAME_LENGTH + 1 ], int length )
{
#ifdef DEBUG
    cerr << "-> insert, to_be_inserted = " << to_be_inserted << endl;
    cerr << "length = " << length << endl;
    cerr << " name array = " << endl;
    for ( int i = 0; i < length; ++i ) cerr << " " << name[ i ] << endl;
#endif
    // BUG FIX: 30 Jul 1999 − make copy of to_be_inserted in case it’s an
    //          alias for something in the name array. Use the copy from here
    //          on.
    char to_be_inserted_copy[ NAME_LENGTH ];
    strcpy( to_be_inserted_copy, to_be_inserted );

    int position = length − 1;
    while ( position >= 0 && strcmp( name[ position ], to_be_inserted_copy ) > 0 )
    {
#ifdef DEBUG
        cerr << "loop: position = " << position
             << ", name[" << position << "] = " << name[ position ]
             << ", to_be_inserted_copy = " << to_be_inserted_copy << endl;
#endif
        strcpy( name[ position + 1 ], name[ position ] );
        --position;
    }
#ifdef DEBUG
    cerr << "out of loop: position = " << position
         << ", name[" << position + 1 << "] = " << name[ position + 1 ]
         << ", to_be_inserted_copy = " << to_be_inserted_copy << endl;
#endif
    strcpy( name[ position + 1 ], to_be_inserted_copy );
#ifdef DEBUG
    cerr << "[end of insert] name array = " << endl;
    for ( int i = 0; i <= length; ++i ) cerr << " " << name[ i ] << endl;
    cerr << "<-insert" << endl;
#endif
}

Figure 1.19: Correcting aliasing of strings in the insert function.
as a sequence of steps) until the level of detail is sufficient to convert the design into a coded solution.

**Object-oriented Design** - Method of solving a problem by identifying the objects involved and the interactions with the objects which must take place. The objects present in a system can often be inferred from nouns used in the description of the system. There may be more than one instance of a particular type of object. Interactions with the objects, called methods can be inferred from verbs.

**Waterfall model** - Model which represents the software development process as a series of stages which flow into each other. Typical stages in the waterfall model are Requirements Analysis, System Specification, System Design, Detailed Design, Implementation, and Verification.

**Black-box testing** - Testing a piece of software without looking at the code. The software is treated as a black box to which the tester feeds input and verifies that the resulting output is correct.

**White-box testing** - Testing a piece of software by designing test cases based on the code. The software is treated as a white box (sometimes referred to as a glass-box) which be examined and verified by the tester. debugging printout - Print statements inserted in code such that they can be output only during the debugging stage. This is accomplished by making use of preprocessor directives to enclose the debugging code and using the -DDEBUG switch during compilation as shown below:

```c
#ifdef DEBUG
  cerr << "Value of index: " << index << endl;
#endif

g++ -Wall -g -DDEBUG main.cpp
```

1.8 Notes

There are many good books on software design and many recent (not necessarily good) books on object-oriented design. Be wary of authors (or coworkers) who treat a particular design methodology and the tools associated with it as if it were a religion. The purpose of design methodologies is to stimulate thinking about a programming task.

For additional reading about software design and programming in general, the wisdom of the classics is unsurpassed: *Classics in Software Engineering* [You79], *The Psychology of Computer Programming* [Wei71], and *Mythical Man Month* [Bro75] are three enjoyable places to start. One of the best references on object-oriented design is Horstmann's *Practical Object-Oriented Development in C++ and Java* [Hor97].
Chapter 2

Separate Compilation and Makefiles

Large pieces of software are usually maintained in many different source files (files containing the text of the program; those having extensions .cpp and .h). To illustrate the techniques used in developing and maintaining programs that are spread out among many files, we use a simple, small example.

Consider the program in Figure 2.1, which computes the mean and median of a collection of non-negative integers (for example, test grades).

Normally, the definition of the sort function would be placed below main in the same file. However, we can also put it into a file by itself, as illustrated in Figure 2.2.

We can compile both files using the command:

% g++ averages.cpp sort.cpp

You might be tempted to think that the compiler simply combines the two source files into one and translates the result to machine-executable form. Not so. Each source file is compiled separately to produce a corresponding object file. The object file contains the source code translated into machine language and (this is important) enough information to allow the file to be linked with others to form a program. So g++ creates two separate object files and then links them together to create the executable file a.out. After you type ./a.out (or simply a.out in most environments), the executable file is loaded into memory and begins execution.

What kinds of errors can occur?

- The function declaration (or prototype) for sort at the top of averages.cpp can fail to match the call of sort in main. This error will be discovered at compile time (when the compiler looks for syntax errors and translates the source program into an object file) because the mismatched items appear in the same file. For example, had we swapped the order of the arguments in the call to make it

  sort(number_of_scores, score);

35
//: averages.cpp − program to compute mean and median of test scores

#include<iostream.h>
#include"sort.h"

const int MAX_SCORES = 100; // maximum number of scores that this
// program can handle

int main()
{
    int total = 0; // sum of all scores
    int number_of_scores = 0;
    int score[MAX_SCORES]; // array to hold all the scores

    // gather the data...
    bool done = false;
    while( !done ) {
        cout << "Enter score, −1 to end: ";
        int current_score;
        cin >> current_score;
        done = ( −1 == current_score );
        if( !done ) {
            score[number_of_scores++] = current_score;
            total += current_score;
            if( MAX_SCORES <= number_of_scores ) {
                cout << "Sorry, can't handle any more scores!" << endl;
                done = true;
            }
        }
    }

    // compute the statistics...
    if( 0 == number_of_scores ) {
        cout << "No scores were entered." << endl;
    } else {
        float mean = float(total) / number_of_scores;
        sort(score, number_of_scores);
        float median = float(score[number_of_scores / 2]);
        if( 0 == number_of_scores % 2 ) { // number of scores is even
            median = (score[number_of_scores / 2 − 1] + score[number_of_scores / 2]) / 2.0;
        }
        cout << "Mean = " << mean << ", median = " << median << endl;
    }
    return 0;
}

Figure 2.1: The file averages.cpp.
we would get the following cryptic warning:

averages.cpp: In function ‘int main()’:
averages.cpp:42: warning: passing ‘int’ to argument 1 of ‘sort(int *, int)’
lacks a cast
averages.cpp:42: warning: passing ‘int *’ to argument 2 of ‘sort(int *, int)’
lacks a cast

The reason is that int and int * (the compiler’s designation for an array of integers; we’ll learn more about the * later) are not the same type (this should really be an error rather than a warning, but a lot of legacy C programs rely on the fact that C allowed conversion between these two types; don’t ask...).

- We might leave out the prototype altogether, thinking that the compiler will figure it out if we compile sort.cpp first. No such luck. Remember, compilation of sort.cpp is completely independent from compilation of averages.cpp, even if the files are listed on the same command line. The message we get then is:

averages.cpp: In function ‘int main()’:
averages.cpp:42: warning: implicit declaration of function ‘int sort(...)’
A euphemism for "you failed to declare your function prototype" (again, should be an error, but C allowed ... you know the story).

• Finally, we might forget to include sort.cpp in the command line with averages.cpp, or we might define the sort function in sort.cpp with arguments in the wrong order. Either infraction gets us:

Undefined first referenced symbol in file
sort_.FPii /var/tmp/cca0061C2.o
ld: fatal: Symbol referencing errors. No output written to a.out

which looks totally incomprehensible unless we know what’s going on. The error message comes from the linker ld (really stands for "loader" because it prepares the executable for loading). The linker’s main job is to resolve all function definitions. When there are separately compiled object files, the call of a function might appear in a different file from its definition, as is the case with our sort function.

From each object file, the linker can create a list of the functions it calls (or, using more general terminology, the symbols it references — separately compiled files can share variables, too) and another list of the functions whose definitions it provides. If every call has a matching definition, then the linker has resolved the corresponding symbols and the pieces of the program can be stitched together correctly. How does the linker figure out whether the prototype of the sort function in averages.cpp (whose object file the compiler has stored as /var/tmp/cca0061C2.o) matches the one in sort.cpp? The g++ compiler uses a convention known as name mangling to attach prototype information to the function name. So sort_.FPii means the sort function (F) with arguments int array (or pointer to int, Pi) and int(i). Other compilers use name mangling as well, but the encoding is different - name mangling has not been standardized, unfortunately, so programs compiled with different compilers cannot be linked properly. The Unix utility, nm, can be used to print the symbol table (name list) for an object file:

csc% nm averages.o
000002b8 t Letext
0000080 r MAX_SQRES
U _Q_qtod
00000000 ? __FRAME_BEGIN__
U __ls__7ostreamPCc
U __ls__7ostreamPFR7ostream_R7ostream
U __ls__7ostreamf
U __rs__7istreamRi
U __throw
U cin
U cout
U endl__FR7ostream
00000000 T main
U sort__.FPii
2.1. Why Bother?

Do you have to learn all the ins and outs of name mangling to decipher error messages from the linker? Fortunately not. There’s a tool called `c++filt` that translates the mangled name to a readable prototype for you. Here’s what to do:

```bash
% add gnu
% echo sort__FPi | c++filt
sort(int *, int)
```

Messages from the linker should now seem a lot less cryptic. They almost always mean that a function called in one file is not defined anywhere or has a mismatching prototype. You can use `c++filt` to find out what the prototype of the call was. You might also get a message about a symbol being "multiply defined". That means the linker found more than one definition among the separately-compiled object files (for example, you might have put a copy of the definition at the end of `averages.cpp` as well as in `sort.cpp`).

2.1 Why bother?

With these new linker errors to worry about, separately-compiled source files might seem like a lot of unnecessary trouble. But there are clear advantages and the benefit obtained by separate compilation increases with the size of the program. Since typical software projects deal with millions of lines of source code, separate compilation is a necessity rather than a luxury. Here are the most compelling reasons to maintain separate source files for different parts of the same program.

1. **Administration.** It is easier to administer large programs where different people work on different parts if only one person or group is responsible for editing each source file. Small, separately-compiled source files make this possible. Modification dates supplied by the system can then be used to determine when each part of the program was last changed and system access control can determine who could have made the change.

2. **Efficiency.** In very large programs, significant time can be saved by recompiling only those program parts that have been changed. The way to set up for this in our example would be:

```bash
% g++ -c averages.cpp
% g++ -c sort.cpp
% g++ averages.o sort.o
```

The `-c` switch stands for "compile only" (don’t attempt to link) and causes the compiler to store the object file as `basename.o` (when the corresponding source is `basename.cpp`) instead of in a temporary directory. When it is given a list of object files, as in the final command, `g++` skips the compilation step and invokes the linker directly.

Later, if we make changes only to `averages.cpp` we could say:

```bash
% g++ -c averages.cpp
% g++ averages.o sort.o
```
and save the time it takes to compile sort.cpp. Not a big deal here, but it could make a lot of difference to someone repeatedly testing and debugging a 20-line function in a million-line program.

3. Reuse. The sort function is certainly useful in contexts other than this little program. Isolating it in a file by itself allows us to use it with other programs easily—without having to edit it into a different source file or even recompile it; we can simply link sort.o into any program that uses it.

2.2 Header files

The errors resulting from inconsistent usage of the sort function can all be discovered at compile time (rather than link time) if the prototype for the function is put into a header file that can be shared among all source files that call sort. The header file, called sort.h, is shown in Figure 2.3.

In place of the prototype at the top of averages.cpp we put:

```c
#include "sort.h"
```

This tells the C preprocessor to insert the text of sort.h here before compiling the source file. You’re already accustomed to this mechanism for definitions and functions supplied by system libraries such as iostream. The only difference is the use of quote marks "" to mean ‘look in the current directory’ instead of angle brackets <> to mean ‘look in the system directory’ (<usr/include>.

There are other C preprocessor commands, all denoted by a # in column 1 of the source file. For example, the sequence

```c
#ifndef SORT_H
#define SORT_H
// actual contents of the sort.h file
#endif
```

is a standard way to keep the contents of a header file from being inserted more than once. The preprocessor inserts the text between #ifndef SORT_H (meaning "if the symbol SORT_H is not defined") and #endif only if SORT_H is undefined. When the text is inserted, the preprocessor reads the #define SORT_H command which prevents future insertions.

Use of the header file has two advantages: (a) a client programmer (programmer who wants to use the function in his/her program) can consult the header file to find out how to call the function; so it serves as documentation; and (b) any misuse of the function in the client program (program
written by the client programmer, averages.cpp in this example) will be discovered by the compiler rather than by the linker. The compiler still has access to line numbers in the source file and can give more complete information about the incorrect call.

To summarize, a separate source file (called, for example, file.cpp) containing definitions of one or more functions is usually accompanied by a corresponding header (called file.h) containing the prototype(s) and comments that are of use to a potential client programmer. A client program, i.e. another source file that needs to call the functions, can do so by (a) putting the directive #include"file.h" at the beginning (the # must be in column 1), and (b) linking in file.o.

The diagram in Figure 2.4 summarizes the compilation and linking process for our example.

2.3 Makefiles

When large programs are composed of separately-compiled source files an automatic method for keeping track of what needs to be compiled or recompiled becomes a useful tool for several reasons.

- It may be hard to remember what files need to be compiled and linked to create (build) the executable version of the program. Accidentally leaving out something results in Undefined symbol errors from the linker, which force you to remember where the appropriate functions were supposed to have been defined.

- When only a few changes have been made since the last build, it's inconvenient to have to remember exactly which files were affected.

- Much useful software is available as source code that can be compiled and linked on a variety of different hardware platforms. Shipment of such software must include instructions for compiling it as a first step in installation.

The tool that addresses these issues is a system utility (program) called make and the data file that it reads is traditionally named makefile or Makefile.

Let's begin assuming the executable program in our example is to be named something other than the generic a.out, say averages to go with averages.cpp. If a makefile has been set up correctly we can say: % make averages. The make program consults the makefile (first it looks for a file called makefile in the current directory, then one called Makefile; with the -f switch the user can also specify a different file name, but this is rarely done). Figure 2.5 illustrates how the makefile for this example might be constructed.

The make program first looks for the target labelled averages in the makefile. Anything followed by a colon in the makefile is a target. Targets are usually names of machine-generated files and the makefile gives instructions on how to generate them. The colon is followed by a list of files (make calls them dependencies) that must be up to date before averages is linked or relinked. The next line, which must start with a TAB character (be careful: some editors, such as Ted, convert tabs to spaces), specifies the linking command. The -o switch after g++ specifies that the executable output file should be stored as the file named immediately after the -o.

Warning: the -o switch causes the previous contents of whatever file is named after it to be lost; a common, but costly, mistake is to say

% g++ -o averages.cpp sort.cpp
when you meant

% g++ -o averages averages.cpp sort.cpp
Figure 2.4: Separate compilation and linking.
The latter compiles and links the two source files to create an executable named averages; the former destroys averages.cpp!

Any sequence of commands can be put after the line containing the target and its dependencies, each on a separate line beginning with a tab. For example, you could have make copy your program to your bin directory after it is done linking:

```
averages: averages.o sort.o
   g++ -o averages averages.o sort.o # this line must begin with a TAB!
averages.o: averages.cpp sort.h
   g++ -c averages.cpp
sort.o: sort.cpp sort.h
   g++ -c sort.cpp
```

HOME is an environment variable that is preset to your home directory.

**What does it mean for a target to be up to date.** The dependencies of a target are either targets themselves and/or files in the directory from which make is being run. To be up to date, a target has to have a later modification time than any of the files (targets) in its dependency list, and any dependencies that are targets themselves have to be up to date with respect to their dependencies, and so on, until we get to a target whose dependencies are all up to date. Suppose we have the following situation:

```
csc% ls -l
total 18
-rwx------ 1 mfms ncsu  404708 Aug  7 11:13 averages
-rw------- 1 mfms ncsu  1385 Aug  7 11:09 averages.cpp
-rw------- 1 mfms ncsu  2236 Aug  7 11:09 averages.o
-rw------- 1 mfms ncsu  323 Aug  7 11:07 makefile
-rw------- 1 mfms ncsu  804 Aug  7 11:11 sort.cpp
-rw------- 1 mfms ncsu  254 Aug  7 14:52 sort.h
-rw------- 1 mfms ncsu  820 Aug  7 11:12 sort.o
```

The file sort.h, which is in the dependency list for target averages.o, has a later modification time than averages.o. Thus, make will execute the command for averages.o, which, in turn, will cause averages.o to have a later time than the target averages. Then make will execute the command for averages.
csc% make averages
 g++ -c averages.cpp
 g++ -o averages averages.o sort.o

When repeatedly recompiling averages.cpp to deal with syntax errors, you can say `make averages.o` instead of `g++ -c averages.cpp`; the effect is the same. In the absence of a specific target `make` always creates the first target in the makefile as a default, so `make` would be equivalent to `make averages` in our example. Sometimes a makefile is used to create a whole collection of programs by beginning with a dummy target. For example:

```
# makefile for creating executable1, executable2, and executable3

all: executable1 executable2 executable3

executable1: # dependencies and commands for executable1 follow here...

# ... and the dependencies and commands for the other 2 are listed here
```

The target `all` does not correspond to any file, nor is any command executed to “create” it. Its only purpose is to serve as the default (first) target; its dependency list causes all the other targets to be created (since a nonexistent target can never be regarded as up to date). The user can simply type `make` and all three executables will be created without the user having to know what they are. This is especially useful for delivering software in the form of source code.

Another common use of a dummy target is to provide a convenient mechanism for issuing a command without having to remember all the details. For example, suppose a programming assignment has you writing several programs (say the executables are `prog1`, `prog2`, and `prog3`). When you’re done working and ready to submit, you no longer need the executables and the `.o` files in your directory (and they could take up a lot of space). You can say `make clean`, assuming your makefile has a dummy target that looks like:

```
clean:
  rm *.o prog1 prog2 prog3
```

This time there are no dependencies - the command is always executed. You can further customize this to get rid of backup files created by your editor.

### 2.4 Makefile errors

The two most likely error messages you’ll get from the `make` program are:

1. Don’t know how to make target, as in

```
make: Fatal error: Don’t know how to make target ‘averages.o’
```

This means `averages.o` appears in a dependency list but it’s neither a target in its own right nor does it appear as a file anywhere in the current directory. This particular error occurred when the information associated with target `averages.o` was deleted from the example makefile.
2.5 Other features of makefiles

When there are many source files, it seems tedious to repeat essentially the same compilation command for each one. A useful shortcut is to define a generic target:

```make
.SUFFIXES: .cpp
.cpp.o:
g++ -c $*.cpp
```

The above incantation keeps us from having to list the command

```make
g++ -c averages.cpp
```

for the target `averages.o` and

```make
g++ -c sort.cpp
```

for the target `sort.o`, and any other commands that create object files from C++ source files. In a large program this could save a lot of repetition.

When a target is not up to date, `make` first looks for commands listed under that target. If there are no such commands, it looks at the list of *suffixes* ( endings of file names, usually but not necessarily of the form `.x`, where `x` is 1 to 3 symbols long) to see if the suffix of the target appears before the suffix of a dependency (having a later modification date). In our example, a target with a `.o` suffix causes `make` to find the `.cpp` suffix later on the list. This occurred because we added `.cpp` as a “dependency” to the `.SUFFIXES` pseudo-target, putting it at the end of the list of suffixes. Standard suffixes, such as `.o` and `.c` are on the list by default, as is `.cc`, the default Sun OS suffix for C++ files (if we had used `.cc` as our suffix for the source file, there would have been no need to add anything - it would already be after `.o` on the list).

Next, `make` looks for a *generic target* of the form `dependSuffixTargetSuffix` and executes the commands listed under that. In our example the `.cpp.o` generic target specifies the command(s) executed to create a `.o` file from the corresponding `.cpp` file. The `$*` in the command stands for the original target with its suffix (`.o`) removed.

**To summarize by way of our example.** Suppose `make` finds that `averages.cpp` has a later modification date than `averages.o` and there is no command listed under the target `averages.o`. Scanning the list of suffixes, it finds `.cpp` after `.o` and executes the command for the `.cpp.o` target, replacing `$*` with `averages` (which is `averages.o` with the `.o` suffix removed).

Another shortcut is to use a *macro* to define a repeatedly used string. Suppose, in a makefile for several programs, every program links in `sort.o`, `search.o`, and `table.o`. We could do the following:
FUNCTIONS = sort.o search.o table.o

all: prog1 prog2 prog3

prog1: prog1.o $(FUNCTIONS)
   g++ -o prog1 prog1.o $(FUNCTIONS)

prog2: prog2.o $(FUNCTIONS)
   g++ -o prog2 prog2.o $(FUNCTIONS)

prog3: prog3.o $(FUNCTIONS)
   g++ -o prog3 prog3.o $(FUNCTIONS)

A line containing an = defines a macro and a $ followed by the macro name causes the macro to be expanded (replaced by its definition). The parentheses are absolutely necessary. Without them, make applies the $ only to the first character after it. Macros are also sometimes used to put important information that a user might want to change at the top of the makefile. For example:

# please edit the following lines to suit your preferences
COMPIlER = g++
FLAGS = -c -g -Wall

# ... later on
.cpp.o:
   $(COMPIlER) $(FLAGS) $*.cpp

This allows the user to choose a different C++ compiler and/or different compiler flags without having to peruse the whole makefile (note: -g means generate code for using the debugger and -Wall means issue all warnings; a more extensive list of compiler options is given later).

The code for the example used here, along with a working makefile can be found in the subdirectory Examples/Separate Compilation of the course locker. More information about make and makefiles is available in the man page for make (do man make).

2.6 A “GNU” syntax for makefiles

A more free form syntax for makefiles that avoids the problems associated with having to have tabs at the beginnings of lines has been introduced by GNU. The version of make on Sun workstations (and Linux) automatically recognizes the new syntax. On other EOS/UNITY platforms you can add gnu. In any case, the old syntax described earlier will still work. In the new syntax each target is described by a single line in which commands are separated from dependencies and from each other by semicolons. A backslash (\) at the end of a line means that the next line is a continuation of the dependencies and commands of the same target. The GNU syntax is illustrated in Figure 2.6.
### 2.7. Compiler options

Some of the more commonly used compiler options (flags) are described in Figure 2.7. More compiler options and variations on these options are listed in the man page for g++ (do man g++).

<table>
<thead>
<tr>
<th>#: makefile for averages program using GNU syntax (and incorporating some # of the features mentioned earlier).</th>
</tr>
</thead>
<tbody>
<tr>
<td>averages: averages.o sort.o; g++ -o averages averages.o sort.o;</td>
</tr>
<tr>
<td>cp averages $(HOME)/bin</td>
</tr>
<tr>
<td>averages.o: averages.cpp sort.h; g++ -c averages.cpp</td>
</tr>
<tr>
<td>sort.o: sort.cpp sort.h; g++ -c sort.cpp</td>
</tr>
<tr>
<td>clean: ; rm *.o averages</td>
</tr>
</tbody>
</table>

Figure 2.6: The **makefile** using GNU syntax.

#### 2.8. Key Concepts

**Preprocessing** - Processing done before a source file is compiled. It may involve the **inclusion** of other files in the file being compiled, the **definition** of symbolic constants and macros, the **conditional compilation** of program code, and the **conditional execution** of preprocessor directives. Preprocessing time errors may involve any of the previously mentioned items.

**Compilation** - Process of checking the syntax of a source file and translating it into an object file. Errors which may occur at **compile time** include syntax errors and function calls which do not match function prototypes.

**Linking** - Process of combining object files to create an executable program. Errors which may occur at **link time** include undefined or multiply defined symbols.

**Runtime errors** - Errors which may occur during the execution of a program include division by zero, logic errors, segmentation faults, and bus errors.

**Code reuse** - Code reuse is fostered by isolating a useful function or set of functions in its own file. That way a number of programs can make use of it.
<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>-c</code></td>
<td>compile only, don't link; store output in <code>file.o</code>, where <code>file.cpp</code> is the source.</td>
</tr>
<tr>
<td><code>-Dsymbol</code></td>
<td>Define the given symbol (give it the value 1); usually used to interact with the C preprocessor; for example, <code>-DTRACE</code> might be used to tell the compiler to include trace printout statements that occur between the directives <code>#ifdef TRACE</code> and <code>#endif</code> (this is a standard way to get lots of useful printout while debugging; the main advantage is that the printout can be omitted during normal execution if you simply recompile without the <code>-DTRACE</code> option - no re-editing required).</td>
</tr>
<tr>
<td><code>-g</code></td>
<td>add code required by a debugger such as gdb.</td>
</tr>
<tr>
<td><code>-I</code> directory</td>
<td>Look for system Include (header) files in the given directory; for example, programs that interact with the X11 window system have to be compiled with <code>-I/usr/X11R6/include</code>.</td>
</tr>
<tr>
<td><code>-l</code> library</td>
<td>include the given library when linking; for example, <code>-lm</code> is used to include the math library, <code>-lx11</code> is used to include the X11 library.</td>
</tr>
<tr>
<td><code>-L</code> directory</td>
<td>Look for Library files in the given directory during linking; the library files have names of the form <code>libname.a</code>, where <code>name</code> is the name of the library; for example, programs that interact with the X11 window system have to be linked on some systems using <code>-L/usr/X11R6/lib</code> (the default is <code>/usr/lib</code>).</td>
</tr>
<tr>
<td><code>-o</code> file</td>
<td>store output in the file <code>file</code> (there is a space between the <code>-o</code> and the file name).</td>
</tr>
<tr>
<td><code>-O</code></td>
<td>Optimize the object (and executable) code so that it runs faster; typical optimizations include taking instructions out of loops if they don't affect the computation in the loop body, and storing values in registers instead of variables whenever possible. You can not use this option and <code>-g</code> at the same time.</td>
</tr>
<tr>
<td><code>-WwarningOption</code></td>
<td>Enable specific kinds of compiler Warnings; use <code>-Wall</code> to turn on all kinds of warnings (this is the best strategy).</td>
</tr>
</tbody>
</table>

Figure 2.7: Options for the g++ compiler.
Chapter 3

Finite-State Machines

A finite-state machine (FSM) is an abstract model of a system (physical, biological, mechanical, electronic, or software). Key components are

- a finite number of states which represent the internal “memory” of the system by implicitly storing information about what has happened before.
- transitions which represent the “response” of the system to its environment. Transitions depend upon the current state of the machine as well as the current input and often result in a change of state.

Consider, for example, the use of an FSM to model an old-time soda machine that dispenses soda for 30 cents (see Figure 3.1). The possible inputs to the machine are n - nickel, d - dime, q - quarter, s - select soda. A bubble diagram can be used to represent the FSM. The states of the machine are designated by circles, each labeled with the amount of money that has been deposited so far. State 00 is designated as the start or initial state by the incoming arrow. States which represent a total input of 30 or more cents are considered final states and are designated by double circles. The transitions from state to state are shown as arcs (lines with arrows). Each transition is labeled with the input that caused it.

If a person puts a nickel into the machine followed by a dime followed by a quarter, the FSM would transition from state 00 to state 05 to state 15 to final state 40. At that point, he or she could select a soda (and hopefully receive 10 cents in change).

A finite-state machine can also be used to model a simple automated teller machine (ATM). In Figure 3.2 each transition is labeled with an input coming from the user (such as insert card) or a central database (such as PIN OK versus bad PIN). The level of abstraction in this example is much greater than that of the soda machine. Each of the transitions may, in actuality, involve a number of complex steps which are not shown on the diagram. The finite-state machine is being used as a system design tool, rather than as a detail-oriented description of the ATM’s operation.

Syntax of Real Numbers. Another application of finite-state machines is as a notation for the precise description of a programming language syntax. For example, real constants in Pascal are described in English by a leading programming text as follows:

A real number in PASCAL is written as a string of digits containing a decimal point. There must be at least one digit before and after the decimal point.
Figure 3.1: A finite-state machine model of a soda dispenser.

Figure 3.2: Finite-state machine showing the operation of an automated teller.
3.1 TEXT PROCESSING WITH FSM’S

Real data may also be represented using PASCAL scientific notation. A real data item written in scientific notation consists of a sign followed by a real number, followed by the letter E, another sign and an integer (+ signs may be omitted).

The English description is imprecise. Are the strings .9 and 9. valid, or do you have to say 0.9 and 9.0, respectively?

The finite-state machine in Figure 3.3 gives a clear answer to this question. A valid real constant in Pascal is any string that leads the FSM from the start state to a final (double-circled) state.

For the first string .9, whose first character is a decimal point ., the FSM stops in state 0 because only a digit or a sign will allow it to move to another state. The second string 9. causes the FSM to go to state 2 based on the first character 9 and to state 3 based on the second character which is a decimal point. It cannot go beyond state 3 since there are no more characters in the string. Since neither string causes the FSM to get to either final state 4 or 7, neither is valid. However, using the FSM with the strings 0.9 and 9.0 proves that these are indeed valid Pascal real constants. Note that .9 and 9. are valid in the C++ language.

3.1 Text Processing with FSM’s

Finite-state machines are often used in text processing. A simple example is a string search that takes place in an editor or in the grep utility, which is used to search a file for a particular pattern. The grep utility takes a string or regular expression and converts it to a finite-state machine before doing a search. To simplify this scenario, suppose a file consists of a’s and b’s only and the search is for the string “abba”. The corresponding finite-state machine is in Figure 3.4 — the double circle indicates a successful conclusion to the search.

If, for example, this FSM were used to locate the string “abba” in a file containing “aababbbaba...”, it would move from state to state as shown in Figure 3.5. The states listed above are the states of the machine after the corresponding input. When the final state 4 is reached, the search is successful.

For now we will use FSM’s to design text-processing programs. What follows is a simple case study to show how an FSM design can be developed and translated directly into a program.

Consider the following specifications:

- A word is a maximal sequence of upper- and lower-case letters. For example, in the string Hello, Judy!, the sequence Judy! is not a word because it includes ! (not a letter), and the
sequence Judy is not a word because it’s not maximal (it does not contain all of the contiguous letters). However, Judy is a word. In the string Goodbye now, Goodbye is considered a word (by our definition) because it is maximal and contains only letters, even though it should be spelled Goodbye.

- wc is the word count, initially 0
- lc is the line count, initially 0
- cc is the character count, initially 0

The FSM in Figure 3.6 both clarifies the definition of a word (which might have been ambiguous without the examples) and gives a design for a program that counts words, lines, and characters in a text file. Each transition is labeled with an input character (or “other”, which means any character that does not appear as a label on a transition from that state) and a corresponding action. For example, the transition labelled “A-Za-z ++wc, ++cc” from state 0 to state 1 means "when the current state is 0 and the next input character is a letter, enter state 1 and increment both wc and cc". In state 0, the FSM remembers that we’re not currently in the middle of a word, while state 1 remembers that we are.
3.1. TEXT PROCESSING WITH FSM’S

#include <iostream.h>    // cin, cout
#include <ctype.h>       // isalpha()
int main()
{
    int wc = 0, lc = 0, cc = 0;
    char ch;
    int state = 0;
    while ( cin.get(ch) ) {
        switch (state) {
            case 0:
                if ( 'n' == ch ) {
                    ++lc; ++cc;
                } else if ( isalpha(ch) ) {
                    state = 1;
                    ++wc; ++cc;
                } else
                    ++cc;
                break;
            case 1:
                if ( 'n' == ch ) {
                    state = 0;
                    ++lc; ++cc;
                } else if ( isalpha(ch) )
                ++cc;
                else
                    state = 0;
                    ++cc;
        }
    }
    cout << lc << "\n" << wc << "\n" << cc << endl;
}

Figure 3.7: A word count program based on the finite-state machine in Figure 3.6.

Another way to depict an FSM uses a transition table instead of a bubble diagram. Each row of the table corresponds to a state and each column to a possible input symbol (or category of input symbols). The transition table is useful for a comment at the beginning of a file that implements a finite-state automaton. The table version for the word counting FSM is given below (as it would appear in a comment). Each table entry shows the new state the machine would enter, based on its current state and current input, and the action(s) that would be taken.

<table>
<thead>
<tr>
<th>STATE</th>
<th>A-Za-z</th>
<th>\n</th>
<th>other</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1 : ++wc,++cc</td>
<td>0 : ++lc,++cc</td>
<td>0 : ++cc</td>
</tr>
<tr>
<td>1</td>
<td>1 : ++cc</td>
<td>0 : ++lc,++cc</td>
<td>0 : ++cc</td>
</tr>
</tbody>
</table>

A standard idiom for translating an FSM into a program is the while-switch idiom. A while
Figure 3.8: Finite-state machine for printing words that are not in C++ comments.

The simplified program is illustrated in Figure 3.10.

This simplified version of the program is included solely to point out that it may not be obvious that a particular piece of code is actually implementing a finite-state machine. For now, it is best to stick with the more straightforward original implementation.

A slightly more complicated use of a finite-state machine is to output each of the words contained in a C++ source file while excluding any that occur in comments. For this example, a word will be defined as any string of alphabetic characters. Comments are defined as strings that begin with the characters // and end with the new line character 'n'. The FSM for this task is shown in table form (to be used as a comment in front of the program) in Figure 3.8 and the program is in Figure 3.9.

In these two examples and all subsequent ones, the outer while loop inputs and processes exactly one character per iteration. This makes it easy to relate the FSM to the code for debugging and modification.
#include <iostream.h>
#include <ctype.h>     // isalpha

int main()
{
    enum { IGNORE, IN_WORD, ONE_SLASH, IN_COMMENT };  
    int state = IGNORE;
    char buf[512];
    int pos;
    char ch;
    while (cin.get(ch)) {
        switch (state) {
        case IGNORE:
            if (isalpha(ch)) {
                pos = 0;
                buf[pos++] = ch;
                state = IN_WORD;
            }  
            else if ('/' == ch )
                state = ONE_SLASH;
            break;
        case IN_WORD:
            if (isalpha(ch)) {
                buf[pos++] = ch;
                state = IN_WORD;
            }  
            else {
                buf[pos] = '\0';
                cout << buf << endl;
                if ('/' == ch)
                    state = ONE_SLASH;
                else
                    state = IGNORE;
            }  
            break;
        case ONE_SLASH:
            if ('/' == ch)
                state = IN_COMMENT;
            else if (isalpha(ch)) {
                pos = 0;
                buf[pos++] = ch;
                state = IN_WORD;
            }  
            else
                state = IGNORE;
            break;
        case IN_COMMENT:
            if ('
' == ch)
                state = IGNORE;
            break;
        }
    }
}

Figure 3.9: The program for the FSM in Figure 3.8.
3.2 Horner’s Rule

A finite-state machine can also be used to convert an ASCII string of characters representing a real number to its actual numerical value. It performs the same operation as the `atof` (ASCII to floating point number) function and uses an algorithm known as Horner’s Rule.

Horner’s Rule is actually a method for evaluating polynomials. Suppose, for example, that you wanted to evaluate \(5x^4 - 3x^3 + x^2 - 4x - 7\) at the point \(x = 3\). The slow way is to compute all the powers of 3 up to \(3^4 \ (3, 9, 27, 81)\) and then multiply them by the appropriate coefficients. A faster way is to rewrite the polynomial as follows:

\[
5x^4 - 3x^3 + x^2 - 4x - 7 = (((0 + 5)x - 3)x + 1)x - 4)x - 7
\]

The 5 ends up being multiplied by \(x\) 4 times, the \(-3\) 3 times, and so on. Now the result is much easier to compute (by hand or by calculator): Start with 0 (always) plus 5 is 5 times \(x(3)\) is 15 minus 3 is 12 times \(x(3)\) is 36 plus 1 is 37 times \(x(3)\) is 111 minus 4 is 107 times \(x(3)\) is 321 minus 7 is 314.

Recall that a decimal integer is actually a polynomial evaluated at \(x = 10\). For example, 1092 is \(1x^3 + 0x^2 + 9x + 2\) evaluated at \(x = 10\). This observation (combined with Horner’s rule) allows us to compute the value of an integer on the fly as we read it from left to right: 0 plus 1 is 1 times 10 is 10 plus 0 is 10 times 10 is 100 plus 9 is 109 times 10 is 1090 plus 2 is 1092. As each digit is read, the correct value of the integer so far has been accumulated.

Horner’s rule can also be used to “read” numbers in any base: just let \(x = B\), where \(B\) is the base. This leads to a nifty trick for evaluating binary numbers: write the binary number, leaving space between bits; start above the leftmost bit with the value 0; cross off the bit below or above
3.3 Finite-state machine based on Horner’s Rule

Two additional details are needed to complete the process of converting a string to a floating-point number. First, to convert a digit (one of the characters ‘0’, ‘1’, ‘2’, ‘3’, ‘4’, ‘5’, ‘6’, ‘7’, ‘8’, ‘9’) to its value, you need only subtract the value of the digit ‘0’ (trust me). Second, digits to the right of the decimal point can be handled by multiplying them by the appropriate negative power of 10 (or positive power of 0.1) — there is no direct way to apply Horner’s rule here.

The FSM is illustrated in Figure 3.11 — the letters shown in the FSM stand for the following:

c - current character
s - sign of the number
v - value of the number
p - power

Note that this FSM assumes that the string contains a valid floating point number that starts with an optional + or -, has at least one digit, an optional decimal point, and any number (including 0) of digits before and after the decimal point.

Like the atof function, a value of 0 is returned if an invalid string is encountered.

The actual code that implements the finite-state machine of Figure 3.11 is in Figure 3.13. As with the program that outputs words not in comments, a table version of the FSM, to be used as a comment for the program, is shown in Figure 3.12.

Since we’re dealing with a string stored in an array (instead of an input stream), the method for getting the next character and the corresponding loop control are modified accordingly.

3.4 Key Concepts

Finite-state machine - Model of the interaction between a system and its environment.
Input symbols are written in **bold** above the transitions, Symbol classes (such as *digit*) are written in *italic.*

Actions performed by the program are in **courier** below the transitions.

The variable *c* denotes the input character.

Figure 3.11: Finite-state machine for converting a string to a number (using Horner’s rule).

---

```
// TABLE form of FSM:
//
// state  |  + or −  |     .    |    digit    |  other  |
//---------|-----------|------------|-------------|---------|
// START   | INTEGER   | DECIMAL    | INTEGER     | ERROR   |
//---------|-----------|------------|-------------|---------|
// INTEGER | ERROR     | DECIMAL    | INTEGER     | ERROR   |
//---------|-----------|------------|-------------|---------|
// DECIMAL | ERROR     | ERROR      | DECIMAL     | ERROR   |
```

Figure 3.12: Table form of the FSM for converting a string to a number.
```cpp
#include <ctype.h>  // isdigit
#include <iostream.h>  // cerr

double string_to_number(const char* string)
{
    double sign = 1;  // sign of the number (either 1 or -1)
    double value = 0;  // current value of the number
    double power = 0.1;  // current power of 10 for digits after decimal point
    int i = 0;
    enum {START, INTEGER, DECIMAL, ERROR} state = START;
    char ch;  // current character in string
    while ((state != ERROR) && (ch = string[i++])) {
        switch (state) {
            case START:
                if ('.' == ch) {
                    state = DECIMAL;
                } else if ('-' == ch) {  // sign of the number
                    sign = -1.0;
                    state = INTEGER;
                } else if ('+' == ch) {
                    state = INTEGER;
                } else if (isdigit(ch)) {
                    value = ch - '0';
                    state = INTEGER;
                } else {
                    state = ERROR;
                }
                break;
            case INTEGER:
                if ('.' == ch) {
                    state = DECIMAL;
                } else if (isdigit(ch)) {
                    value = 10.0 * value + (ch - '0');
                } else {
                    value = 0.0;
                    state = ERROR;
                }
                break;
            case DECIMAL:
                if (isdigit(ch)) {
                    value += power * (ch - '0');
                    power /= 10.0;
                } else {
                    value = 0.0;
                    state = ERROR;
                }
                break;
            default: cerr << "Should not be here!" << endl;
        }
    }
    return sign * value;
}
```

Figure 3.13: Program for converting strings to numbers, based on the FSM in Figure 3.11.
State - Representation of the internal memory of the system being modeled by the FSM. A state
remembers what has occurred so far. In addition to the FSM state, there may be variables
that remember other details. The designer has to use judgment to decide what to model with
a FSM state and what to leave as a variable.

Transition - Action that occurs when an event in the environment causes the system to change
state (either the FSM state or variables).

Bubble diagram - Graphical representation of a FSM which uses labeled circles for the states and
arcs (lines with arrows) for the transitions. The arcs are labeled with the input that caused
the transition along with any action that occurs during the transition.

Transition table - Tabular representation of a FSM in which each row corresponds to a state and
each column corresponds to an input symbol or category. The table entries contains the next
state a FSM would enter based on the state and input along with any action(s) that would be
taken.

Text-processing - A FSM can be used to model text-processing. Each transition corresponds to
a single input character. This is extremely important: otherwise the FSM cannot be translated
mechanically into a program.

While-switch idiom - Method for mechanically translating a FSM to a program. All input to the
program should occur in exactly one place in the program (at the top of the while loop).

Horner's Rule - Method for converting a sequence of digits in any base to an integer. It is based
on the evaluation of polynomials.

There are many excellent books about finite-state machines and their uses. The early chapters of
texts by Martin [Mar97, chapter 3] and Wood [Woo87, chapter 2] make a good springboard
for further exploration.
Chapter 4

Classes

4.1 Background

A class is a means of encapsulating (tying together) data and functions associated with the data. A class can be used to instantiate (create) one or more objects of the class type. A class is simply a description of a particular type of object and what it can do (and what can be done to it).

Consider a class named Animal. An Animal can have a name, a kind, a color, and a weight. It can eat, walk, and sleep. A black 20 pound dog named Blackie, a yellow 1 pound bird named Goldie, and a white 7 pound cat named SnowBall are all objects of type Animal. They can all eat, walk, and sleep. Note that the Animal class simply describes the characteristics of an Animal and what it can do.

4.2 The Date class

The first C++ class we will be looking at in detail is the Date class. The data associated with a particular Date is its month and day. We can create a Date object. We can get the month and day for a particular Date object and we can set the month and day for a Date object. We can also ask a Date object to output its month and day.

The definition of the Date class is split into two different files. The first file, Date.h, which appears in Figure 4.1, is the header file for the class. Someone (generally referred to as a client) who wants to use the Date class in a program would only need to look at and use this file which defines the interface to the class. The term client may also be used to refer to another program part that uses the class. The functions for the class are defined in another file, Date.cpp, which appears in Figure 4.2 and is known as the implementation file. The actual definition of the Date class begins with the line:

    class Date {

The body of the Date class is enclosed in curly braces({}). NOTE the semi-colon(;) following the closing } — this is very important. The functions that are available to a user (client) of the Date class are defined in the section denoted by the label public. The data for the class, which is not
```cpp
//: Date.h − Date class header file
//  Suzanne Balik, 8 Jul 1999

#ifndef DATE_H
#define DATE_H

#include <fstream.h>

class Date {
friend ostream& operator<<(ostream& out, const Date& d);

public:
    Date(int month = 1, int day = 1);
    void setDate(int month, int day);
    int getMonth() const;
    int getDay() const;

private:
    int myMonth;
    int myDay;
};

#endif
```

Figure 4.1: Date class header file.

accessible to the client except through the public functions, is defined in the section denoted by the label `private:`.

**Class Constructor.** The constructor for the class is declared by the line:

`Date(int month = 1, int day = 1);`

Notice that the constructor for the class has the same name as the class and does not have a return type. It can be used to instantiate objects of type `Date` as follows:

`Date laborDay(9,6);
Date birthday(10,25);`

The constructor for the `Date` class gives default values for both the month and the day. If a client does not specify a month and a day for a particular `Date` object, the month and day will be set to 1. When instantiating a `Date` object with no arguments, the parentheses are omitted as follows:

`Date today;  //The month and day are set to 1`

If only one argument is given, it is used to initialize the month, which is the first argument, and the day is set to 1.

`Date tomorrow(9);  //The month is set to 9; the day is set to 1`

Since it is possible to create a `Date` object without any arguments, the `Date` constructor functions as a **null constructor** for the class. In order to declare an array of objects for a class, a null constructor must be used. An array of `Date` objects can be defined as follows:
4.2. The Date Class

Figure 4.2: Date class implementation file.

```cpp
//: Date.cpp − Date class implementation file
//  Suzanne Balik, 8 Jul 1999

#include "Date"

ostream& operator<<(ostream& out, const Date& d)
{
    out << d.myMonth << ' − ' << d.myDay << endl;
    return out;
}

Date::Date(int month, int day)
{
    myMonth = month;
    myDay = day;
}

void Date::setDate(int month, int day)
{
    myMonth = month;
    myDay = day;
}

int Date::getMonth() const
{
    return myMonth;
}

int Date::getDay() const
{
    return myDay;
}
```

Date holidays[10]; //The month and day for all holidays are set to 1

Class Functions. The function:

```cpp
void setDate(int month, int day);
```

can be used to set the values of the private data variables, `myMonth` and `myDay`, and is referred to as a setter function. It can be used as follows:

```cpp
today.setDate(9,16);
```

The functions:

```cpp
int getMonth() const;
int getDay() const;
```

return the value of `myMonth` and `myDay` respectively and are known as getter functions. Note the use of `const` with these functions. Declaring a class function `const` means that the function cannot
change the values of the private class data. A const function is the only type of function that can be called on a class object that has been declared as const. For example, only the getMonth and getDay functions could be called on the following object:

    const Date valentinesDay(2, 14);

Attempting to change the month or day of valentinesDay by using one of the setter functions would cause a compiler error.

The function:

    friend ostream& operator<<(ostream& out, const Date& d);

is an overloaded output (stream-insertion) operator. It is defined as a friend\(^1\) of the Date class which means that it has access to the private data of the class. This function could be invoked as follows:

    cout << laborDay;

When the compiler sees this expression, it will generate the following function call:

    operator<<(cout, laborDay);

This function cannot be defined as a member function of the Date class because the class object appears on the right-hand side of the << operator. Since it must have access to the private class data, the only choice is to define it as a friend function of the class. Normally defining functions as a friend of a class is highly discouraged because it violates the object-oriented principle of information hiding by giving the function access to the private member data of the class.

**Preprocessor Directives.** Note that the Date class definition is preceded by two preprocessor directives:

    #ifndef DATE_H
    #define DATE_H

and followed by the following preprocessor directive #endif. This tells the preprocessor to include the code following #define DATE_H only if the symbol DATE_H has not been previously defined. This is important when writing larger programs where the header file, Date.h, may be included more than once. Note that by convention, the symbolic name DATE_H is formed by capitalizing the name of the header file and replacing the period with an underscore.

The implementation file, Date.cpp, contains the definitions of all of the Date class member functions as well as the overloaded output operator, which is a friend function of the class. The Date.cpp file contains the line:

    #include "Date.h"

which tells the preprocessor to include the code contained in the Date.h file.

**Date Member Functions.** Each of the Date member functions is preceded by the name of the class, Date, and the binary scope resolution operator(::), for example:

    Date::getMonth();

\(^1\)The use of friend functions when not absolutely necessary is discouraged.
### 4.2. THE DATE CLASS

This identifies the function as a member of the Date class. The data and member functions of a class are said to belong to that class’s **scope**. This means that any class function can access the class data as well as call another class function directly. The class constructor is defined as follows:

```cpp
Date::Date(int month, int day)
{
    myMonth = month;
    myDay = day;
}
```

It simply initializes the private data members, `myMonth` and `myDay`. It is called automatically whenever a Date object is instantiated. Other code can be included in the constructor. For example, to output a message whenever the Date constructor is used to create a new object, you could add the line:

```cpp
cerr << "Date constructor: month - " << myMonth << " day - " << myDay;
```

to the constructor definition. The code for the other class functions should be self-explanatory.

**Overloaded Output Operator.**

The code for the overloaded output operator is as follows:

```cpp
ostream& operator<<(ostream& out, const Date& d)
{
    out << d.myMonth << '-' << d.myDay << endl;
    return out;
}
```

Notice that a constant reference(`const Date&`) to a Date object is passed to this function. The `const` qualifier specifies that the function cannot modify the class data. Since it is not a class function and does not have class scope, it cannot access the data directly. Instead it must refer to it using the dot operator(,) as `d.myMonth` and `d.myDay`. Notice also that the overloaded output operator returns a reference to the output stream with the statement:

```cpp
return out;
```

This enables a phenomenon known as **cascading**, which means that the overloaded output operator can be used in conjunction with other calls to the output operator (overloaded or otherwise). Thus a statement like:

```cpp
cout << "Labor Day: " << laborDay << " New Year’s Day: " << newYearsDay;
```

is legal because each call to `operator<<` returns the output stream, `cout`, which is then used on the left-hand side of the next call to `operator<<`. This can be illustrated as follows:

```cpp
cout << "Labor Day: " << laborDay << " New Year’s Day: " << newYearsDay;
    cout << laborDay << " New Year’s Day: " << newYearsDay;
        cout << " New Year’s Day: " << newYearsDay;
            cout << newYearsDay;
```
A simple main function that makes use of the Date class appears in Figure 4.3. The line

```cpp
#include "Date.h"
```

provides access to the Date class. A makefile that can be used to compile and link the main function with the Date class to create an executable program called dates appears in Figure 4.4. The output from the dates program is shown in Figure 4.5.

### 4.3 The DateList class

It is possible for a class to include as data one or more objects of another class. This is known as a “has a” relationship because one class has an object of another class. It is also referred to as composition since one class is composed of objects of another class. We will now consider a DateList class which has an array of Date objects. The header and implementation files for the DateList class are shown in Figures 4.6 and 4.7.

The definition

```cpp
Date myDates[MAX_SIZE];
```
4.3. **THE DATELIST CLASS**

---

```
# makefile for dates program
# Suzanne Balik, 14 Dec 1999

dates: main.o Date.o
    g++ -o dates -g main.o Date.o

main.o: main.cpp Date.h
    g++ -c -Wall -g main.cpp

Date.o: Date.cpp Date.h
    g++ -c -Wall -g Date.cpp

clean:
    rm *.o dates
```

**Figure 4.4:** Makefile for dates program.

---

csc% dates
Valentine’s Day: 2-14
Today: 9-16
New Year’s Day: 1-1
Labor Day: 9-6
Birthday: 3-15
Today: 10-4
Test Dates:
9-15
11-4

**Figure 4.5:** Output from dates program.
//: DateList.h − Date List class header file
// Suzanne Balik, 9 Jul 1999

#ifndef DATE_LIST_H
#define DATE_LIST_H

#include "Date.h"

class DateList {

friend ostream& operator<<(ostream& out, const DateList& dl);

// PRE: we're at the beginning of a line
// POST: each date in the list has been printed on a separate line and
//       we're at the beginning of the line following the last date

public:
    DateList();
    bool isFull() { if (myNumberOfDates != MAX_SIZE) return false;
                else return true; }

    bool add(int month, int day);
    bool add(Date date);

private:
    enum {MAX_SIZE = 10};
    Date myDates[MAX_SIZE];
    int myNumberOfDates;
};
#endif

Figure 4.6: Date List class header file.
#include "DateList.h"

ostream& operator<<(ostream& out, const DateList& dl) {
    for (int i = 0; i < dl.myNumberOfDates; i++)
        out << dl.myDates[i] << endl;
    return out;
}

DateList::DateList() : myNumberOfDates(0) {};

bool DateList::add(int month, int day) {
    if (myNumberOfDates != MAX_SIZE) {
        myDates[myNumberOfDates].setDate(month, day);
        myNumberOfDates++;
        return true;
    }
    else
        return false;
}

bool DateList::add(Date date) {
    if (myNumberOfDates != MAX_SIZE) {
        myDates[myNumberOfDates] = date;
        myNumberOfDates++;
        return true;
    }
    else
        return false;
}
creates an array of Date objects. An anonymous enumeration

```cpp
enum{MAX_SIZE = 10};
```

is used to set the size of the array.

**Inline Functions.** The function

```cpp
bool isFull() { if (myNumberOfDates != MAX_SIZE) return false;
  else return true; }
```

is defined in the class definition which makes it an **inline** function. This means that if the compiler chooses to do so, it will insert a copy of the **actual code** for the function anywhere that the function is called. This saves the overhead of the function call, but makes the program larger.

**Initialization List.** The constructor

```cpp
DateList::DateList() : myNumberOfDates(0) {
}
```

uses an **initialization list** to set the data member, `myNumberOfDates`, to 0 rather than doing it in the body of the constructor. An initialization list starts with a colon (:) and is followed by one or more of the data members separated by commas. The initialization value for each data member is enclosed in parentheses. Using an initialization list for the Date class constructor would result in:

```cpp
Date::Date(int month = 1, int day = 1) : myMonth(month), myDay(day) {
}
```

**Overloaded Functions.** The `add` function for the DateList class is **overloaded**:

```cpp
bool add(int month, int day);
bool add(Date date);
```

This means that the same name has been used for two functions in the same **scope**. This is permissible if the functions have different numbers and/or types of parameters or if one function is declared as a `const` function and the other is not. The combination of the function’s name and its parameter types is known as its **signature**. The compiler uses a process known as **name mangling** to distinguish between the functions. It encodes the function identifier with the number and types of its parameters. These are the mangled names for the functions above:

```cpp
add_SDateListii - ii for the two integer parameters
add_SDateListG4Date - Date for the Date parameter
```

A program named `c++filt` can be used to decipher mangled names (see Appendix B).

**Overloaded Assignment Operator/Copy Constructor.** The compiler automatically provides an overloaded assignment operator for a class which is what makes possible this assignment statement in the second version of the `add` function:

```cpp
myDates[myNumberOfDates] = date;
```

The compiler also provides a copy constructor for the class which is used if a class object is passed in a function call:
4.4. STRUCTURES VS. CLASSES

//: main.cpp − main function to use Date and DateList classes
// Suzanne Balik, 9 Jul 1999
#include <iostream.h>
#include "Date.h"
#include "DateList.h"

int main()
{
    // create a DateList instance called 'fallHolidays'
    DateList fallHolidays;
    // recall that a DateList instance has (a private member that is) an
    // array of Date instances called 'myDates' (with size MAX_SIZE)
    // Add two dates to 'fallHolidays' using the first prototype of the
    // 'add' function, namely 'add(int month, int day)'
    fallHolidays.add(9,6);
    fallHolidays.add(10,11);
    // Add another date using the second prototype of 'add', namely
    // 'add(Date date)'; the expression 'Date(10,12)' creates a temporary
    // instance of class Date (which is copied into 'fallHolidays')
    fallHolidays.add(Date(10,12));
    // Create an instance of class Date called 'thanksgiving' and add it to
    // 'fallHolidays' (using the second prototype)
    Date thanksgiving(11,25);
    fallHolidays.add(thanksgiving);
    // add another temporary instance of Date to 'fallHolidays'
    fallHolidays.add(Date(11,26));
    cout << "Fall Holidays: " << endl << fallHolidays;
}

Figure 4.8: main function for datelist program.

void printTheDate(Date d, ostream& out);
.
.
.
Date laborDay(9,6);
printTheDate(laborDay, cout);

The datelist Program. A main function that uses the DateList class together with a makefile and output appear in Figures 4.8, 4.9, and 4.10.

4.4 Structures vs. Classes

In the C++ programming language the structure data type, commonly referred to as a struct, is identical to the class data type with one exception. In a class, the default member accessibility is private while in a struct, it is public. In other words, in a struct, any data or member functions
# makefile for datelist program
# Suzanne Balik, 10 Jun 1999

datelist: main.o Date.o DateList.o
  g++ -o datelist -g main.o Date.o DateList.o

main.o: main.cpp Date.h DateList.h
  g++ -c -Wall -g main.cpp

Date.o: Date.cpp Date.h
  g++ -c -Wall -g Date.cpp

DateList.o: DateList.cpp DateList.h Date.h
  g++ -c -Wall -g DateList.cpp

clean:
  rm *.o datelist

---

csc% datelist
Fall Holidays:
  9-6
  10-11
  10-12
  11-25
  11-26

---

Figure 4.9: makefile for datelist program.

Figure 4.10: Output from datelist program.
not specifically declared as private will be considered to be public members of the struct. In a class, the opposite is true – data or member functions not specifically declared as public will default to private. A struct is declared using the `struct` keyword and can be used identically to a class:

```c
struct Time {
    public:
        Time(int hour, int min);
        int getHour();
        int getMin();
    private:
        int myHour;
        int myMin;
};
```

In the C programming language, a struct can be used solely as a means to aggregate data. It cannot contain a constructor to initialize the data or functions to operate on the data. Data members of a struct are initialized or modified directly using the dot operator (.) (or the arrow operator which will be introduced in Chapter 5).

```c
struct Time {
    int hour;
    int min;
}
```

```c
Time lunchTime;
lunchTime.hour = 12;
lunchTime.min = 30;
```

### 4.5 Key Concepts

**Class** - Data type consisting of data members and functions that operate on the data. The name of a class can be used in place of a type to declare *instances* (objects) of the class.

**Public class members** - Class data members and functions that are accessible to clients of the class as well as class functions.

**Private class members** - Class data members and functions that are accessible to class functions but not to clients of the class. Class data is typically defined as private.

**Constructor** - Class function called whenever a new instance is created to ensure proper initialization of the class data.

**Null (default) constructor** - Class constructor which does not require any arguments. A null constructor must be used to create an array of class objects.

**Function definition syntax** - A class member function definition begins with the function’s return type followed by the class name and the function name separated by the binary scope resolution operator (::) followed by the argument list. For example, `void Date::setDate(int month, int day)`. This is then followed by the body of the function.
**Calling member functions** - Class member functions are called for a particular class instance by following the object’s name with a period and the function name and argument list. For example, `myBirthday.setDate(4, 21);

**const class functions** - Class functions which are guaranteed *not* to change the data of a class object. They are declared as `const` by following the function’s argument list with the keyword, `const`, e.g., `int getMonth() const`. They are the *only* type of function which can be called on a `const` class object, e.g., `const Date independenceDay(7, 4);

**Initialization list** - Method of initializing data members when an object is created. An initialization list is defined by following the constructor name with a colon (:) and then the names of each of the data members to be initialized separated by commas. Each data member is followed by its initial value enclosed in parentheses. The initial value for a data member can be any expression using constants and arguments to the constructor. For example, `Date::Date(int month = 1, int day = 1) : myMonth(month), myDay(day)`

`DateList::DateList() : myNumberOfDates(0)`

**Preprocessor directives (ifndef, define, endif)** - Used in the class header file to ensure that the class is declared *once and only once* in any client code that includes the class. For example,

```
#ifndef DATE_H
#define DATE_H

class Date {

};
#endif
```
Chapter 5

Pointers and Dynamic Memory Management

Written by Dan DuVarney, Fall 1997, based on handwritten notes from Matt Stallmann and Chapter 9 of the Perry/Levin textbook. The “postmaster” analogy came from a discussion (Dan had) with Rick Klevans.

5.1 What are Pointers?

- Each data object has a unique address in the computer’s memory where it is stored.
- Addresses are just numbers. They typically are consecutive integers 0, 1, 2, ... up to some large value (maybe 2^32).
- A pointer is a data object which contains the address of some other data object.
- In the case where x contains the address of y, programmers typically say: “x points to y.”

For example, consider the layout of memory shown in Figure 5.1. A diagram of the memory layout is shown in Figure 5.2. The memory layout might be created by executing the following C++ code:

```cpp
int y = 112;
int* x = &y;
char* z = "hello";
```

5.2 Pointers in C++: * and &

In C++, there are three uses for * and &. Each has a separate meaning as

- a Type constructor
- a unary operator, and
- a binary operator.

Figure 5.3 summarizes this.
<table>
<thead>
<tr>
<th>Address</th>
<th>Value</th>
<th>Size</th>
<th>Name</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>000</td>
<td>112</td>
<td>4</td>
<td>not stored</td>
<td>y</td>
</tr>
<tr>
<td>104</td>
<td>100</td>
<td>4</td>
<td>x</td>
<td>pointer to int</td>
</tr>
<tr>
<td>107</td>
<td>112</td>
<td>4</td>
<td>z</td>
<td>pointer to char</td>
</tr>
<tr>
<td>112-117</td>
<td>unnamed</td>
<td>char</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Figure 5.1: An example of a memory layout.

Figure 5.2: Memory diagram for the layout presented in Figure 5.1.
### 5.3 Aliases

If `x` points to `y`, then we say that `*x` is an alias for `y`:

```c
float  y = 5.0;
float* x = &y;

*x  *= 2.0;
```

The last line changes `y`’s value to 10.

A **reference type** can also be used to create an alias, and it’s sneaky because no `*` or `&` are used except in the type declaration:

```c
float  y = 5.0;
float& x = y;

x  *= 2.0;
```

Again, the last line changes `y`’s value to 10.

### 5.4 Dynamic Memory

- In C/C++, the runtime library maintains a pool of memory called the **heap**.
- Any subroutine can borrow (**allocate**) memory from the heap via the **new** operator
- All borrowed memory should eventually be returned (**deallocated**) via the **delete** operator

Example: Making a copy of a string in dynamically allocated memory

```c
char * my_strdup( const char * s )
{
    int length = strlen(s) + 1;
    char * s2 = new char[ length ];
    for( int i = 0; i < length; ++i ) s2[ i ] = s[ i ];
    return s2;
}
```
```cpp
#include <iostream.h>

char *
strdup(const char *s)
{
    char *s2 = new char [strlen(s)+1];
    char *dst = s2;

    while (*dst++ = *s++)
    {
    }

    return s2;
}

int
main(int argc, char *argv[])
{
    char *x1 = strdup("once allocated");
    delete [] x1;
    char *x2 = strdup("different string");
    cout << "x1=" << x1 << endl;
    cout << "x2=" << x2 << endl;
}
```

Figure 5.4: A program that causes a dangling pointer.

You can now make a copy of a string as follows:

```cpp
char * myCopy = strdup("this is a string.");
```

Remember, `strdup` returned a pointer to memory which was allocated from the heap. The memory should be returned to the heap as soon as it's no longer needed:

```cpp
delete [] myCopy;
```

Pitfalls of Dynamic Memory Usage.

- **Memory Leak.** Something is allocated, but never deallocated. Did you ever notice programs that tend to get slower and slower (and maybe eventually crash) as you use them? They probably have a memory leak.

- **Dangling Pointer.** You keep using a pointer to a piece of memory that has been deallocated.

- **Corrupted Heap.** The heap was damaged by deallocating something that wasn't allocated.

An example of a program that causes a dangling pointer is shown in Figure 5.4. With `g++` on EOS, you get the following output:

```cpp
x1 =
x2 = different string
```
To avoid the above problem, after every delete, set the pointer to NULL:

```c
delete [] x1;
x1 = NULL;
```

What is **NULL**?

- **NULL** is a special address which isn’t used by the compiler (typically address 0).
- Hence, there’s no way &x will return NULL for any variable x, since the compiler won’t store anything at address NULL.

By forcing `x1` to be **NULL**, You get the following output with g++ on EOS:

```c
x1 = (null)
x2 = different string
```

**NULL Trivia**

- Dereferencing a NULL pointer usually causes a segmentation fault with g++ on a Sun workstation.
- A fairly common bug in programs is that they assume you get 0 when you dereference a NULL pointer. Such code actually works on some machines, even though it’s wrong.

**Q:** Does assigning NULL to pointers after every delete guarantee there will be no dangling pointers? **A:** Unfortunately, No. Consider:

```c
char *x3 = x1;
...
delete [] x3;
...
cout << x1;
```

Since `x3` pointed to the same chunk of memory as `x1`, when we deleted `x3` (deallocated memory pointed to by `x3`, we also deleted `x1`!

## 5.5 Heap Manager

How do **new** and **delete** actually work?

- The C++ runtime library includes a **heap manager** (subroutines responsible for maintaining the heap).
- The heap manager does **bookkeeping** to keep track of which parts of the heap have been loaned out for usage (allocated), and which parts are still **free** (deallocated).
- Every time a call to **new** is made, the heap manager:
  1. searches the free part of the heap for a chunk of memory that is big enough
  2. makes a record of the fact that the chunk of memory is now **allocated**
  3. returns the starting address of the allocated chunk as a result.
• When you call `delete`, the heap manager updates its records to note that the block is free.

**Contract Between Programmer and Heap Manager.** There is an implicit contract between you, the programmer, and the heap manager. The contract has 3 clauses. Violating each clause causes a different bug:

<table>
<thead>
<tr>
<th>Clause</th>
<th>Penalty for Violation</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. You must eventually return all the memory you borrow.</td>
<td>Memory Leak</td>
</tr>
<tr>
<td>2. You must stop using memory as soon as you return it.</td>
<td>Dangling Pointers</td>
</tr>
<tr>
<td>3. You can’t return memory which you didn’t borrow in the first place (including returning the same chunk of memory twice in a row).</td>
<td>Corrupted Heap</td>
</tr>
</tbody>
</table>

**Heap Manager as Postmaster.** One way to think of the heap manager is as a postmaster:

• The heap is like a bunch of P.O. boxes.

• When you call `new`, you are asking to reserve a P.O. box. The postmaster gives you a P.O. Box Number to use.

• When you call `delete`, you give the postmaster a P.O. box number. The postmaster assumes that you no longer are using the P.O. box, so another customer can use it.

• Unfortunately, there are no locks on the P.O. Boxes. Everyone is on the honor system: they are only supposed to use P.O. boxes that they have reserved.

• Obviously, this system is vulnerable to mistakes. People can accidently (or purposefully) take each other’s mail. Figuring out who took your mail can be extremely difficult.

C++ manages its heap in a manner similar to the above post office. As a result, the heap is vulnerable to programming mistakes, and finding the source of a bug can be very difficult! The benefit gained is that the postmaster doesn’t have to do much work, so memory allocation is fast.

You should now be aware of three different ways to allocate memory. These are illustrated in Figure 5.5.

**Automatic Storage and Dangling Pointers.** Returning a pointer to a local variable is another way to get a dangling pointer. Consider the program in Figure 5.6. With g++, you get the following output:

```
s2 = eat my lunch
```

**Corrupted Heap.** The heap can be corrupted when

• A **dangling pointer** is deleted

• A pointer to an object in **static** or **automatic** memory is deleted

**Why is a corrupted heap possible?**

• In order to save time, the heap manager assumes that anything you delete was properly allocated from the heap (very little checking is done).
### 5.5. Heap Manager

<table>
<thead>
<tr>
<th>Allocation Technique</th>
<th>How data is allocated</th>
<th>Lifetime of Data</th>
</tr>
</thead>
<tbody>
<tr>
<td>Static</td>
<td>Global variables, things explicitly tagged with the <code>static</code> keyword.</td>
<td>Until the program halts</td>
</tr>
<tr>
<td>Automatic</td>
<td>Local variables inside functions (unless the <code>static</code> keyword is used).</td>
<td>Until the program exits the block containing the variable.</td>
</tr>
<tr>
<td>Dynamic</td>
<td>Pointer variables assigned the result of a new operation</td>
<td>Until <code>delete</code> is called</td>
</tr>
</tbody>
</table>

Figure 5.5: Three different kinds of memory allocation.

```c
#include <iostream.h>
#include <string.h>

char *
bad_strdup(const char *s)
{
    char s2[10000];
    strncpy(s2, s, 10000);
    return s2;
}

int
main(int argc, char *argv[])
{
    char *s2 = bad_strdup("this is a test");
    char *s1 = bad_strdup("eat my lunch");
    cout << "s2 = " << s2 << endl;
}
```

Figure 5.6: A dangling pointer that results from automatic memory deallocation.
Policies to Avoid Dangling Pointers and Memory Leaks. We can avoid dangling pointers and memory leaks by following these rules:

1. Functions that return pointers should return pointers to dynamically allocated objects.
2. Any function which receives a pointer from a function should delete the pointer or return the pointer as a result.
3. Avoid assigning one pointer to another (make a copy of the object pointed-to instead).
4. Assign NULL to pointers after you delete what they point to.

Are these rules sufficient to guarantee there are no memory leaks or dangling pointers?

5.6 Using Pointers with Class Objects

A pointer can be set to the address of an existing class object. Using the Date class of Chapter 4, we can create a pointer to a Date object:

```cpp
Date date1(2,15);
Date* date1Ptr = &date1;
```

We can also use a pointer to create a dynamically allocated Date object:

```cpp
Date* datePtr = new Date(4,21);
```

If we wanted to call the `getMonth` function using `datePtr`, we could dereference `datePtr` using the dereference operator (`*datePtr`) and then use the dot operator (`.`) to call the function:

```cpp
int month = (*datePtr).getMonth();
```

The parentheses are necessary because the dot operator has a higher precedence than the dereference operator. This means that, without the parentheses, the dot operator would be evaluated before the dereference operator. This notation is rather awkward and can be replaced by the arrow operator (->):

```cpp
int month = datePtr->getMonth();
```

5.7 Classes with Dynamically Allocated Memory

When classes contain dynamically allocated memory, we need to make sure that the memory is handled properly. The code for a Name class, which contains a pointer to a dynamically allocated character string used to store a name, is shown in Figures 5.7 and 5.8. The class constructor allocates new memory:

```cpp
Name::Name(const char* name)
{
    myName = new char[strlen(name)+1];
    strcpy(myName, name);
}
```
Figure 5.7: Name class header file.

The setName function deallocates the old memory associated with the character string before allocating new memory:

```cpp
void
Name::setName(const char* name)
{
    delete [] myName;
    myName = new char[strlen(name)+1];
    strcpy(myName, name);
}
```

This class also contains three very important functions which should be implemented whenever a class contains dynamically allocated memory:

- a destructor
- a copy constructor
- an overloaded assignment operator

If these functions are not implemented for a class, the compiler provides default versions of each. However, the default versions do not handle the dynamically allocated memory properly, which can lead to serious problems.

**Destructor.** A destructor has the same name as the class, but preceded by a tilde (~), e.g. ~Name. It does not have a return type and takes no arguments. It is automatically called:
```cpp
#include "Name.h"

ostream& operator<<(ostream& out, const Name& n) {
    out << n.myName;
    return out;
}

Name::Name(const char* name) {
    myName = new char[strlen(name)+1];
    strcpy(myName, name);
}

Name::~Name() {
    cerr << "Destructor: " << myName << endl;
    delete [] myName;
}

Name::Name(const Name& other) {
    myName = new char[strlen(other.myName)+1];
    strcpy(myName, other.myName);
}

Name& Name::operator=(const Name& rhs) {
    if (this != &rhs) {
        delete [] myName;
        myName = new char[strlen(rhs.myName)+1];
        strcpy(myName, rhs.myName);
    }
    return *this;
}

void Name::setName(const char* name) {
    delete [] myName;
    myName = new char[strlen(name)+1];
    strcpy(myName, name);
}
```

Figure 5.8: Name class implementation file.
5.7. **Classes with Dynamically Allocated Memory**

1. For class objects that go out of scope.

2. For class objects that are deallocated using the `delete` operator.

3. For class objects in existence when a program exits.

The destructor for the Name class deallocates the memory pointed to by `myName`. It also prints a message so that you can tell when it’s been called:

```cpp
Name::Name()
{
    cerr << "Destructor: " << mName << endl;
    delete [] mName;
}
```

The purpose of the destructor is to prevent memory leaks when an object goes out of existence. As a general rule, an object that allocates memory when created must be responsible for deallocating that memory when it is destroyed.

**Copy Constructor.** The **copy constructor** can be used to create a new class object which is a copy of an existing class object:

```cpp
Name boy1("Bobby");
Name boy2(boy1);
```

The Name class object, `boy1`, is created using the class constructor. The object, `boy2`, is created using the copy constructor with `boy1` as an argument, which results in `boy2` having the same name as `boy1`. The copy constructor:

```cpp
Name::Name(const Name& other)
{
    mName = new char[strlen(other.mName)+1];
    strcpy(mName, other.mName);
}
```

takes as an argument a constant reference to the “other” Name class object. It does a **deep copy** of the “other” object’s `mName` character string. This means that it allocates new memory and then copies the “other” object’s character string to it. If we had not implemented this function, the compiler would provide a default copy constructor. This copy constructor would do a **shallow copy** of the Name object. It would be **dangerously** implemented as:

```cpp
Name::Name(const Name& other)
{
    mName = other.mName;
}
```

In this situation, both class objects would have pointers to the exact same memory location! In our example, `boy1` and `boy2` both have the name “Bobby”. If `boy2` decides to change his name to “Tommy” by directly overwriting the data (not allowed in the client program, of course, but for illustrative purposes):

```cpp
strcpy( boy2.mName, "Tommy" );
```
boy1's name will become "Tommy" as well. This is because both "boys" share the same memory location for their "names". If the name change is done using the `setName` member function, the first line of that function deallocates `boy2.myName`, thereby making `boy1.myName` a dangling pointer. Figures 5.9 and 5.10 illustrate the difference between deep and shallow copy.

**Overloaded Assignment Operator.** The overloaded assignment operator is used to assign one class object to another:

```cpp
boy1 = boy2;
```

It is similar to the copy constructor, but a little more complicated:

```cpp
Name&
Name::operator=(const Name& rhs)
{
    if (this != &rhs) {
        delete [] myName;
        myName = new char[strlen(rhs.myName)+1];
        strcpy(myName, rhs.myName);
    }
    return *this;
}
```

The object on the right-hand side (`rhs`) of the equal sign is passed as a constant reference, which means that it cannot be modified by the function. The first thing the function does is check whether an object has been assigned to itself (e.g., `boy1 = boy1;`). It does this by comparing the address of the right-hand side object (`&rhs`) to the **this pointer**. The **this pointer** contains the address of **this**
5.8. KEY CONCEPTS

class object, i.e., the one for which the function was called. If they are indeed the same, we don’t want to do any assignment. If they’re not the same, then the memory associated with the character string is deallocated. New memory large enough to hold the “right-hand side” name is allocated and that name is copied to it. A reference to this object (*this) is returned. This enables cascading, e.g.,

```cpp
boy1 = boy2 = boy3;
```

which is implemented by the compiler as:

```cpp
boy1.operator=(boy2.operator=(boy3));
```

A `main` function, which uses each of these functions, and its output appear in Figures 5.11 and 5.12.

**Private Copy Constructor/Overloaded Assignment Operator.** One way to avoid the problem of the compiler supplying default versions of the copy constructor and overloaded assignment operator is to define them as empty private class member functions:

```cpp
private:
    Name(const Name& other) {}  // copy constructor
    Name& operator=(const Name& rhs) {}  // overloaded assignment operator
```

This is sufficient in situations where neither function is needed by a client of the class. This method protects a client who unintentionally calls one of the functions from potentially corrupting the class object’s dynamically allocated data. Figure 5.13 shows the compiler errors that result from attempting to use a privately defined copy constructor or overloaded assignment operator.

### 5.8 Key Concepts

**Address** - Location in the computer’s memory where a particular data object is stored. Addresses are simply numbers. The address of a data object, `x`, is obtained by *preceding* the object name with an ampersand (`&`) as in `&x`.

**Pointer** - Data object which contains the address of another data object. Pointers are *declared* by *following* the data type of the object by an asterisk (`*`). For example,

```cpp
int* intPtr;  // Pointer to an integer
float* floatPtr;  // Pointer to a float
Date* datePtr;  // Pointer to a date object
```

Pointers are *dereferenced* by *preceding* the name of the pointer with an asterisk (`*`). For example,

```cpp
*intPtr;  // The value of the integer pointed to by intPtr;
*floatPtr;  // The value of the float pointed to by floatPtr;
*datePtr;  // A reference to the Date object pointed to by datePtr;
```

**Static memory allocation** - Data objects whose memory is *statically allocated* exist in the program’s data section *throughout* the execution of the program. Global variables and variables explicitly declared as `static` are statically allocated.
//: main.cpp - main driver program for the Name class
// Suzanne Balik, 2 Aug 1999

#include <iostream.h>
#include "Name.h"

int main()
{
    Name boy1("Bobby"); //Uses constructor
    Name boy2(boy1); //Uses copy constructor
    cout << "Boy1: " << boy1 << " Boy2: " << boy2 << endl;
    Name boy3("Jimmy"); //Uses constructor
    boy1 = boy3; //Uses overloaded assignment operator
    cout << "Boy1: " << boy1 << " Boy2: " << boy2 << " Boy3: " << boy3 << endl;
    cout << "Destructor called when objects go out of scope" << endl;
}

Name* girlPtr = new Name("Amy"); //Dynamically allocated Name object
    //Changing name using the pointer
    cout << "Girl: " << *girlPtr << endl; //Dereferencing the pointer
    cout << "Destructor called when object is deallocated" << endl;
    delete girlPtr;

Name dog("Fido");
    cout << "Dog: " << dog << endl;
    cout << "Destructor called when program exits" << endl;
}

---

**Figure 5.11:** name program main function.

```
csc% g++ -o name main.cpp Name.cpp
csc% name
Boy1: Bobby Boy2: Bobby
Boy1: Jimmy Boy2: Bobby Boy3: Jimmy
Destructor called when objects go out of scope
Destructor: Jimmy
Destructor: Bobby
Destructor: Jimmy

Girl: Jill
Destructor called when object is deallocated
Destructor: Jill

Dog: Fido
Destructor called when program exits
Destructor: Fido
```

**Figure 5.12:** name program output.
5.8. KEY CONCEPTS

```cpp
1 //: main.cpp - shows compiler errors when attempting to use private copy
2 // constructor/overloaded assignment operator for Name class
3 // Suzanne Balik, 14 Dec 1999
4
5 #include <iostream.h>
6 #include "Name.h"
7
8 void passByCopy(Name name) {} //Causes constructor to be called
9 void passByReference(Name& name) {} //Does not cause constructor
10 //to be called
11
12 int main()
13 {
14    Name boy1("Bobby");   //Uses constructor
15    Name boy3("Tommy");   //Uses constructor
16    Name boy2(boy1);      //Uses copy constructor
17    passByCopy(boy1);     //Uses copy constructor
18    passByReference(boy1); //Does not use copy constructor
19    boy1 = boy3;          //Uses overloaded assignment operator
20 }
```

csc% g++ main.cpp Name.cpp
main.cpp: In function ‘int main()’:
Name.h:21: ‘Name::Name(const Name&)’ is private
main.cpp:15: within this context
Name.h:21: ‘Name::Name(const Name&)’ is private
main.cpp:16: within this context
Name.h:22: ‘class Name & Name::operator =(const Name&)’ is private
main.cpp:18: within this context

Figure 5.13: Compiler errors using private copy constructor/overloaded assignment operator.
Automatic memory allocation - Data objects whose memory is \textit{automatically allocated} exist in memory from the time the block in which they are declared is entered until it is exited during the program's execution. The memory for automatic variables is allocated from the program's stack.

Dynamic memory allocation - Data objects whose memory is \textit{dynamically allocated} exist in memory from the time they are explicitly allocated using the \texttt{new} operator until they are deallocated using the \texttt{delete} operator. The memory for dynamically allocated variables is allocated from the program's heap.

\textbf{new operator} - Used to dynamically allocate memory for a data object. This memory must be referred to using a pointer. For example,

\begin{verbatim}
int* countPtr = new int;
const int SIZE = 10;
int* array = new int[SIZE]; //array points to an array of 10 int's
int* * arrayPtrs = new int*[SIZE]; //arrayPtrs points to an array of //10 pointers to integers
Date* datePtr = new Date(10,25);
Name* names = new Name[SIZE]; //names points to array of 10 Name objects
\end{verbatim}

\textbf{delete operator} - Used to deallocate dynamically allocated memory for a data object. When deallocating memory for an array, the delete operator must be followed by square brackets ([ ]). For example,

\begin{verbatim}
delete countPtr;
delete [] array;
delete [] arrayPtrs;
delete datePtr;
delete [] names;
\end{verbatim}

\textbf{Dangling pointer} - Pointer which points to memory that has been deallocated. Even though the memory has been deallocated and can be allocated for another data object, the pointer still contains the address of the memory location. One way to avoid this is to set the pointer to 0 after deallocating the memory.

\textbf{Memory leak} - Occurs when a pointer to a dynamically allocated data object is set to point to another object without first deallocating the memory for the original object.

\textbf{Classes with dynamically allocated data} - Classes which contain dynamically allocated data should provide a \texttt{destructor} to deallocate the data when a class object goes out of existence. They should provide a \texttt{copy constructor} which dynamically allocates data for a class object that is created as a copy of an existing class object. They should provide an \texttt{overloaded assignment operator} to deallocate the data of an existing class object and allocate new data when one class object is assigned to another.

\textbf{Shallow copy} - Copying the value of one pointer to another so that both pointers point to the \textit{same} memory location.
5.8. KEY CONCEPTS

Deep copy - Dynamically allocating new memory for a data object and copying the data associated with an existing object to the new memory. Pointers for the two data objects will point to two different memory locations.
CHAPTER 5. POINTERS AND DYNAMIC MEMORY MANAGEMENT
Chapter 6

Linked Lists

6.1 Background

A linked list is a data structure that sequentially links together data items to create a list. The first data item is linked to the second data item which is linked to the third data item and so on. New data items can be linked to the first or last item in the data structure. It is also possible to unlink two data items in the middle of the list and insert a new data item between them. Similarly, it is possible to remove an item from the list by unlinking it from the item(s) that precede(s) and/or follow(s) it. If the item that was removed was located in the middle of the list, the item which preceded it must be re-linked to the item that followed it. Thus, a linked list can shrink and grow dynamically, taking up only as much space as it needs to store all of the data items. It is also essentially unbounded, limited only by the total amount of available memory. Thus, it can grow as large as necessary to hold all of the data items. This is in contrast to the array data structure whose size remains fixed regardless of how many data items it actually contains.

The data items in a linked list must be accessed sequentially, that is, to access the third item in a list, you must start with the first item in the list, move to the second item in the list, and finally reach the third item. Again, this is in contrast to the array data structure whose data items may be accessed directly. This is often referred to as random access since any random item can be accessed with equal efficiency.

There are many different ways that linked lists can be implemented in a given programming language. We will look at a simple way of implementing this data structure in C++. There are also many different ways to perform linked list operations, such as the insertion and removal of data items. The methods presented here are intended to be simple and understandable. As you become more comfortable with linked lists, you may be able to shorten the code for a given operation by combining some of the cases.

6.2 The Node Class

A data item in a linked list is combined with a pointer (or link) to the next item in the list to form a simple structure known as a node. Thus a linked list consists of a number of nodes, each of which
// List.h − List class header file
// Suzanne Balik, 14 Dec 1999
#ifndef LIST_H
#define LIST_H
#include <fstream.h>
class Node;
class List {
friend ostream& operator<< (ostream& out, const List& l);
public:
    List();
    List (const List& other);
    List& operator=(const List& rhs);
    ~List();
    bool isEmpty();
    void add(int info);
    void remove(int info);
private:
    Node* myHead;
};
#endif

Figure 6.1: List class header file.

points to the next node in the list. The last node points to null (or 0) which signifies the end of the list. This is the definition of the Node class that we will be using:

class Node {
public:
    Node (int info, Node* link = 0) : myInfo(info), myLink(link) {}
    int myInfo;
    Node* myLink;
};

It consists of a constructor, whose link argument defaults to 0, and two data members – an integer variable, myInfo, and a node pointer, myLink. The Node class is a self-referential class in that it contains a pointer to another Node class object. Unlike all of the classes we have previously looked at, the data members are declared as public rather than private. This is acceptable because we will hide the Node class declaration in the List class implementation file. Thus only the List class will be able to access the Nodes in the list.
6.2. THE NODE CLASS

```cpp
//: List.cpp - List class implementation file
// Suzanne Balik, 16 Dec 1999

#include "List.h"

class Node {
public:
    Node(int info, Node* link = 0) : myInfo(info), myLink(link) {}
    int myInfo;
    Node* myLink;
};

ostream& operator<<(ostream& out, const List& l) {
    Node* tmp = l.myHead; //Set tmp to point to the head node
    out << "[";
    while(tmp) { //While there more nodes in the list
        out << tmp->myInfo; //Output the info tmp points to
        if(tmp->myLink) //If tmp is followed by another node,
            out << ","; //output a comma
        tmp = tmp->myLink; //Set tmp to point to the next node
    }
    out << "]" << endl;
    return out;
}

static Node* copy(Node* p) //Return a copy of the list which begins
    { //with a node pointed to by p
    Node* head = 0; //head points to the head node of the
    Node* tail = 0; //tail points to the last node of the
    //new list
    while (p) { //While there are still nodes to be copied
        if (!head) //If the new head node has not been created
            head = tail = new Node(p->myInfo); //set head to first new node
        else { //set tail to new node
            tail->myLink = new Node(p->myInfo); //add new node to tail node
            tail = tail->myLink;
        }
        p = p->myLink; //Set p to point to next node in list
    }
    return head;
}
```

Figure 6.2: List class implementation file (first part).
static void deallocate(Node* p) //Deallocate nodes of the list which begins
{                               //with a node pointed to by p
    Node* tmp;
    while (p) {                   //While there are still nodes to deallocate
        tmp = p;                  //Set tmp to p
        p = p->myLink;            //Set p to point to next node
        delete tmp;              //Dealocate tmp
    }
}

List::List() : myHead(0) {}     //Constructor

List::List(const List& other)   //Copy constructor
{                             //Set head node of new list to copy
    myHead = copy(other.myHead); //of other list
}

List::~List()                   //Destructor
{                             //Deallocate the list nodes
    deallocate(myHead);        //Not really necessary, but a good habit
    myHead = 0;
}

List&
List::operator=(const List& rhs) //Overloaded assignment operator
{                               //If this list is not the same list
    if (this != &rhs) {         //as the right-hand side list
        deallocate(myHead);     //Deallocate this list’s nodes
        myHead = copy(rhs.myHead); //Set head node of this list
    }                           //to copy of rhs list
    return *this;               //Return a reference to this list
}

bool List::isEmpty() //Does myHead have a non-zero value?
{                   //If so, the list is not empty.
    if (myHead)        //If not,
        return false; //the list is empty.
    else
        return true;
}

Figure 6.3: List class implementation file (continued).
6.2. THE NODE CLASS

```cpp
void List::add(int info) {
    myHead = new Node(info, myHead); // Add a new node before the head node
}

void List::remove(int info) {
    if (isEmpty()) return; // Is the list empty?
    if (info == myHead->myInfo) { // Does the head node contain the info?
        Node* tmp = myHead; // If so, set tmp to point to it.
        myHead = myHead->myLink; // Move myHead to the next node.
        delete tmp; // Deallocate tmp.
        return; // return.
    }
    Node* prev = myHead; // Set prev to point to the head node
    Node* curr = myHead->myLink; // Set curr to point to the next node
    while (curr) { // While curr is still pointing to a node
        if (curr->myInfo == info) { // Does the current node contain info?
            prev->myLink = curr->myLink; // If so, set the previous node to point
            // to the node after the current node
            delete curr; // Deallocate the current node and
            return; // return
        } else { // If not, move to the next node
            prev = curr; // by setting prev to point to curr
            curr = curr->myLink; // and curr to point to the next node
        }
    }
    return; // The list does not contain info, return.
}
```

Figure 6.4: List class implementation file (continued).
6.3 The List Class

The header and implementation files for the List class appear in Figures 6.1 to 6.4. The only private
data item in the class is a Node* pointer, myHead, which will be used to point to the head (first) node
in the list. If myHead has a value of 0 (null), it means that the list is empty. The List constructor
creates an empty list by setting myHead = 0. The isEmpty function returns true or false signifying
an empty or non-empty list based on the value of myHead. A client program could create a list as
follows:

List theList;

At this point the list would look like this with myHead pointing to null:

\[ 
\text{myHead} 
\]

Executing theList.add(5); would result in the following:

\[ 
\text{myHead} \rightarrow \begin{array}{c}
5 \\
\text{myInfo myLink}
\end{array} 
\]

The myHead member now points to the first (and only) node and the myLink member of that
node now points to null. The statements:

theList.add(2);
theList.add(7);

would result in the following:

\[ 
\text{myHead} \rightarrow \begin{array}{c}
7 \\
2 \\
5 \\
\end{array} 
\]

Each successive node has been added to the front of the list because of the way we have chosen
to implement the add function:

```cpp
void List::add(int info)
{
    myHead = new Node(info, myHead);
}
```
6.3. **The List Class**

Each time a new Node is created, its link field is set to the value of `myHead`, which is the address of the first node in the list (or 0 the first time a new Node is created). The variable, `myHead`, is then set to the address of the new Node. Remember that the `new` operator returns the address of the new memory that has just been allocated.

**Removing an Item.** Removing an item from the list is more complicated. There are four different situations to consider:

1. The list is *empty*.
2. The *head node* contains the item to be removed.
3. A node in the *middle* or at the *end* of the list contains the item.
4. The item *does not appear* in the list.

We will consider each case separately by looking at a pictorial representation of the list removal operation and then the code.

**Case 1.** The list is empty. No picture is needed here — the code trivial:

```java
if (isEmpty()) //Is the list empty?
    return; //If so, return.
```

**Case 2.** The head node contains the item to be removed (in this case, 7). Before removal, suppose the list looks like this:

![Diagram of list with 7 as the head node](image)

The value of `myHead` must be changed to point to the second node. However, we must maintain a pointer to the first node so that it can be deallocated:

![Diagram showing pointer to the first node](image)

After the removal operation the list looks like this:
The code for handling this case is:

```c
if(info == myHead->myInfo) { //Does the head node contain the info?
    Node* tmp = myHead;    //If so, set tmp to point to it.
    myHead = myHead->myLink; //Move myHead to the next node.
    delete tmp;              //Deallocate tmp..
    return;                 //return.
}
```

**Case 3.** The item to be removed is contained by a node in the middle or at the end of the list (in this case, 3). In this situation, we will have to link the node that occurs before the node to be removed to the node that occurs after the node to be removed. To accomplish this, we will have to maintain two pointers as we traverse (move through) the list, one which points to the current node and one which points to the previous node:

We compare the data in the current node (4) to the value that we want to remove (3). Since they are not equal, we move the previous pointer to the current node and the current pointer to the next node:

Since the current node contains an 8 instead of a 3, we move the pointers again:
This time the current node does indeed contain a 3. We must link the previous node to the node following the current node. Then the current node can be deallocated:

The list after the removal looks like this:

The code for accomplishing this is as follows:

```c
Node* prev = myHead;  //Set prev to point to the head node
Node* curr = myHead->myLink;  //Set curr to point to the next node
while(curr) {  //While curr is still pointing to a node
    if (curr->myInfo == info) {  //Does the current node contain info?
        prev->myLink = curr->myLink;  //If so, set the previous node to point
        //to the node after the current node
        delete curr;  //Deallocate the current node and
        return;  //return
    }
    else {  //If not, move to the next node
        prev = curr;  //by setting prev to point to curr
        curr = curr->myLink;  //and curr to point to the next node
    }
}
```
Case 4. The item is not contained in any of the nodes in the list (in this case, 9) In Case 3 above, we would have arrived at situation in which the current pointer is pointing to null and off the end of the list:

![Diagram of list structure]

At this point, we would know that 9 was not contained in the list. This is represented in the code given above with dropping out of the while loop when the pointer, curr, becomes 0. The only thing to do then is simply return:

```cpp
    return;              // The list does not contain info, return.
```

**Helper Functions.** Two helper functions are defined for the use of the List class functions, copy and deallocate. The static keyword ensures that only functions within the List class implementation file, List.cpp, can call the helper functions. The copy function returns a pointer to a list which is a deep copy of another list:

```cpp
static Node* copy(Node* p)  // Return a copy of the list which begins
{                          // with a node pointed to by p
    Node* head = 0;        // head points to the head node of the
    Node* tail = 0;        // of the new list
    while (p) {
        if (!head) {        // If the new head node has not been created
            head = tail = new Node(p->myInfo); // set head to first new node
        } else {
            tail->myLink = new Node(p->myInfo); // add new node to tail node
            tail = tail->myLink;  // set tail to new node
        }
        p = p->myLink;        // Set p to point to next node in list
    }
    return head;
}
```

The deallocate function deallocates all of the nodes of a list:

---

1. This use of the keyword static has nothing to do with static data allocation. The designers of C wanted to avoid adding another keyword, so they used static, which makes no sense when applied to a function. A better keyword might have been local, meaning local to this file.
6.3. THE LIST CLASS

```cpp
static void deallocate(Node* p) //Deallocation nodes of the list which begins
{                           //with a node pointed to by p
    Node* tmp;
    while (p) {             //While there are still nodes to deallocate
        tmp = p;           //Set tmp to p
        p = p->myLink;     //Set p to point to next node
        delete tmp;       //Deallocation tmp
    }
}

Destructor. The destructor for the List class deallocates all of the list nodes:

```cpp
List::List() //Destructor
{          //Deallocation the list nodes
    deallocate(myHead);     //Not really necessary, but a good habit
    myHead = 0;
}
```

Overloaded Output Operator. The overloaded output operator outputs the list in the form: [7, 4, 8, 3, 5]. Since it is a friend function and not a member function of the List class, it must refer to the head node as l.myHead, where l is a constant reference to the List object on the right-hand side of the << operator (e.g., cout << theList):

```cpp
ostream& operator<<(ostream& out, const List& l)
{
    Node* tmp = l.myHead; //Set tmp to point to the head node
    out << '[';
    while(tmp) {          //While there more nodes in the list
        out << tmp->myInfo; //Output the info tmp points to
        if(tmp->myLink)     //If tmp is followed by another node,
            out << ', ',  //output a comma and space
        tmp = tmp->myLink;  //Set tmp to point to the next node
    }
    out << ']' << endl;
    return out;
}
```

Copy Constructor. The copy constructor creates a list as a copy of an existing list:

```cpp
List list1;
list1.add(2);
list2.add(3);
List list2(list1); //Uses copy constructor to create list2
```

It is defined as follows:

```cpp
List::List(const List& other) //Copy constructor
{                          //Set head node of new list to copy
    myHead = copy(other.myHead);
}                         //of other list
```
Overloaded Assignment Operator. The overloaded assignment operator deallocates the memory associated with the nodes of an existing list which appears on the left-hand side of an assignment statement:

```cpp
List list1;
List list2;
.
.
list2 = list1;  //Uses overloaded assignment operator
```

A copy of the list on the right-hand side is then assigned to the list on the left-hand side:

```cpp
List&
List::operator=(const List& rhs)  //Overloaded assignment operator
{
    if (this != &rhs) {  //If this list is not the same list
        //as the right-hand side list
deallocate(myHead);  //Dealocate this list’s nodes
        myHead = copy(rhs.myHead);  //Set head node of this list
    }
    return *this;  //Return a reference to this list
}
```

### 6.4 Ordered Lists

An ordered list is one in which the nodes are maintained in a particular order. The code for an OrderedList Class is given in Figures 6.5, 6.6, and 6.7. This class maintains the nodes in ascending numerical order. Executing the following code:

```cpp
OrderedList theOrderedList;
theOrderedList.insert(4);
theOrderedList.insert(2);
theOrderedList.insert(9);
```

would result in the following ordered list:

```
myHead

2 → 4 → 9
```

You may want to draw diagrams of the `insert` and `remove` functions for this class. In fact, when writing linked list code it is always helpful to

- Consider all of the possible cases that could occur.
6.5. **Linked Lists versus Arrays**

Using an ordered list as an example, we compare a linked list with an array in detail. Suppose the following sequence of operations is done:

OrderedList L; L.insert(3); L.insert(5); L.insert(2);

Figure 6.8(a) shows the result, both in the usual box-and-arrow diagram form and in terms of the layout in memory (the actual addresses of heap objects will, of course be much larger). Assuming there are no other dynamic memory allocations (uses of new) in the program, the locations for the 3 nodes will be contiguous in the order of allocation. But that’s purely coincidental to the structure of the linked list — order is maintained by following links from myHead to the 0 that signals the end.

A subsequent L.remove(3) operation causes one link to change and the memory occupied by the node containing 3 to be deallocated — see Figure 6.8(b). Lastly we do L.insert(1) — the
//: OrderedList.cpp - OrderedList class implementation file
// Suzanne Balik, 10 Jul 1999

#include "OrderedList.h"

class Node {
public:
    Node(int info, Node* link = 0) : myInfo(info), myLink(link) {
        int myInfo;
        Node* myLink;
    }

    ostream& operator<<(ostream& out, const OrderedList& l) {
        Node* tmp = l.myHead;
        out << "[";
        while(tmp) {
            out << tmp->myInfo;
            if(tmp->myLink) out << ",";
            tmp = tmp->myLink;
        }
        out << "]" << endl;
        return out;
    }

    OrderedList::OrderedList() : myHead(0) {}
};

OrderedList::~OrderedList() {
    Node* tmp;
    while (myHead) {
        tmp = myHead;
        myHead = myHead->myLink;
        delete tmp;
    }
}

bool OrderedList::isEmpty() {
    if (myHead)
        return false;
    else
        return true;
}

Figure 6.6: Ordered List class implementation file (first part).
void
OrderedList::insert(int info)
{
    if (isEmpty()) {
        myHead = new Node(info);
        return;
    }
    if (info < myHead->myInfo) {
        myHead = new Node(info, myHead);
        return;
    }
    Node* prev = myHead;
    Node* curr = myHead->myLink;
    while (curr)
    {
        if (info < curr->myInfo) {
            prev->myLink = new Node(info, curr);
            return;
        }
        else {
            prev = curr;
            curr = curr->myLink;
        }
    }
    prev->myLink = new Node(info);
    return;
}

void
OrderedList::remove(int info)
{
    if (isEmpty() || info < myHead->myInfo)
        return;
    if (info == myHead->myInfo) {
        Node* tmp = myHead;
        myHead = myHead->myLink;
        delete tmp;
        return;
    }
    Node* prev = myHead;
    Node* curr = myHead->myLink;
    while (curr && info >= curr->myInfo) {
        if (info == curr->myInfo) {
            prev->myLink = curr->myLink;
            delete curr;
            return;
        }
        else {
            prev = curr;
            curr = curr->myLink;
        }
    }
    return;
}
CHAPTER 6. LINKED LISTS

Figure 6.8: Memory diagrams illustrating operations on an ordered linked list.

Figure 6.9: The operations of Figure 6.8 on a dynamic array.
6.6. STACKS

previously deallocated space is reclaimed for the new node and it is linked to the front, as shown in Figure 6.8(c).

An ordered list can also be implemented as a dynamic array. Private data consists of a pointer to an array of integers (int* myArray) and the current number of items (int myLength). The integers in the array must be stored contiguously — for the linked list it was only coincidence that the nodes appeared contiguously (memory locations between two nodes might have been used by other parts of the program). And, most importantly, the order of appearance in the array is the only order that can be inferred. Figure 6.9(a) shows what the dynamic array implementation looks like after integers 2, 3, and 5 have been inserted (the order of their insertion does not affect the picture). In Figure 6.9(b) we see the result of removing 3 — the 5 is copied over the 3 (it still appears in its original position but is not part of the array because the length has been set to 2). Finally, Figure 6.9(c) shows the result of inserting a 1 — all existing data in the array must be copied to the next higher location before the 1 is put into the first location.

This example illustrates the tradeoffs between linked lists and arrays. Linked lists require extra space for the links but none of the operations require any data to be moved. This can be particularly advantageous when the size of the data is large, for example if each data item were a student record instead of a single integer.

6.6 Stacks

A stack is a data structure in which items are added to the top and removed from the top. This is analogous to a stack of cafeteria trays. Trays that have just been washed are added to the top of the stack. People going through the cafeteria line also take trays from the top of the stack. A stack is an example of a last-in, first-out (LIFO) data structure.

The code for the Stack class is given in Figures 6.10, 6.11, and 6.12. Notice that the pointer variable, myTop, is being used instead of myHead, to signify the top of the stack. In computer science terminology, items are pushed onto the stack and popped off the stack. Therefore the Stack class has a push function which adds an item to the top of the stack and a pop function which removes an item from the top of the stack. The top function simply returns the value of the item on the top of the stack without removing it. Notice the use of assert which will cause the program to abort if a client attempts to use the top function on an empty stack. Executing the following code:

```c
Stack theStack;
theStack.push(2);
theStack.push(5);
theStack.push(3);
theStack.push(1);
theStack.pop();
theStack.pop();
```

would result in the following stack:
Figure 6.10: Stack class header file.

6.7 Queues

A queue is a data structure in which items are added to the rear of the queue and removed from the front of the queue. This is analogous to a cafeteria line (queue) of people waiting to be served. The person at the front of the line is served first. A person enters the line at the rear. A queue is a first-in, first-out (FIFO) data structure.

The code for the Queue class is given in Figures 6.13, 6.14, and 6.15. Notice that the Queue class has two pointer variables, myFront, which points to the node at the front of the list, and myRear which points to the node at the rear of the list. The front function returns the value at the
//: Stack.cpp − Stack class implementation file
// Suzanne Balik, 10 Jul 1999

#include "Stack.h"
#include <cassert.h>

class Node {
public:
    Node(int info, Node* link = 0) : myInfo(info), myLink(link) {}
    int myInfo;
    Node* myLink;
};

ostream& operator<< (ostream& out, const Stack& s)
{
    Node* tmp = s.myTop;
    out << "[";
    while(tmp){
        out << tmp->myInfo;
        if(tmp->myLink)
            out << ",";
        tmp = tmp->myLink;
    }
    out << "]" << endl;
    return out;
}

Stack::Stack() : myTop(0) {}
Stack::~Stack()
{
    Node* tmp;
    while (myTop) {
        tmp = myTop;
        myTop = myTop->myLink;
        delete tmp;
    }
}

bool Stack::isEmpty()
{
    if (myTop)
        return false;
    else
        return true;
}
Figure 6.12: Stack class implementation file (continued).

front of the queue. The addToRear and removeFromFront functions do exactly that. Executing the following code:

```cpp
Queue theQueue;
theQueue.addToRear(5);
theQueue.addToRear(2);
theQueue.addToRear(8);
theQueue.removeFromFront();
theQueue.addToRear(1);
```

would result in the following queue:

---

6.8 Recursion

Thus far, all of our linked list processing has been done iteratively. We used a while loop, which is a repetitive control structure, to traverse the list. Another way of manipulating linked lists is
//: Queue.h - Queue class header file
// Suzanne Balik, 14 Dec 1999

#ifndef QUEUE_H
#define QUEUE_H

#include <fstream.h>

class Node;

class Queue {

friend ostream& operator<< (ostream& out, const Queue& q);

public:
    Queue();
    ~Queue();
    bool isEmpty();
    void addToRear(int info);
    int front();
    void removeFromFront();

private:
    Queue(const Queue& other) {};
    Queue& operator=(const Queue& rhs) {};
    Node* myFront;
    Node* myRear;
};

#endif

Figure 6.13: Queue class header file.
//: Queue.cpp − Queue class implementation file  
// Suzanne Balik, 18 Jun 1999

#include "Queue.h"
#include <assert.h>

class Node {
public:
    Node(int info, Node* link = 0) : myInfo(info), myLink(link) {}
    int myInfo;
    Node* myLink;
};

ostream& operator<<(ostream& out, const Queue& q) {
    Node* tmp = q.myFront;
    out << "[";
    while(tmp) {
        out << tmp->myInfo;
        if(tmp->myLink)
            out << ",";
        tmp = tmp->myLink;
    }
    out << "]" << endl;
    return out;
}

Queue::Queue() : myFront(0), myRear(0) {}

Queue::~Queue() {
    Node* tmp;
    while (myFront) {
        tmp = myFront;
        myFront = myFront->myLink;
        delete tmp;
    }
}

bool Queue::isEmpty() {
    if (myFront)
        return false;
    else
        return true;
}
6.8. RECURSION

```cpp
void Queue::addToRear(int info)
{
    if (isEmpty())
        myFront = myRear = new Node(info);
    else {
        myRear->myLink = new Node(info);
        myRear = myRear->myLink;
    }
}

int Queue::front()
{
    assert (!isEmpty());
    return myFront->myInfo;
}

void Queue::removeFromFront()
{
    if (!isEmpty()) {
        Node* tmp = myFront;
        myFront = myFront->myLink;
        delete tmp;
        if (!myFront)
            myRear = 0;
    }
    return;
}
```

Figure 6.15: Queue class implementation file (continued).
through the use of a *recursive* function, that is a function that calls itself until a base case (stopping condition) is reached.

To illustrate the difference between an iterative and a recursive solution to a programming problem, consider the task of computing the sum of the first \( n \) integers. The iterative approach is to sum the numbers in order from 0 to \( n \), which can be expressed mathematically as

\[
\sum_{i=0}^{n} i
\]

An iterative function could make use of the `for` repetitive control structure to calculate the sum:

```c
int sum(int n)
{
    int sum = 0;
    for(int i = 0; i <= n; i++)
        sum += i;
    return sum;
}
```

A recursive solution to this problem would involve repetitively reducing the problem to a smaller problem until a base case was reached. We would consider the sum of the first \( n \) integers to be the sum of \( n \) plus the sum of the first \( n - 1 \) integers, which is the sum of \( n - 1 \) plus the sum of the first \( n - 2 \) integers and so on, until we reached the sum of the first 0 integers which, of course, is 0. This can be expressed mathematically as:

\[
\sum_{i=0}^{n} i = n + \sum_{i=0}^{n-1} i
\]

where \( \sum_{i=0}^{0} i = 0 \), which is the base case.

Using this approach to calculate the first 4 integers results in

\[
\begin{align*}
    \sum_{i=0}^{4} i &= 4 + \sum_{i=0}^{3} i \\
    &= 4 + 3 + \sum_{i=0}^{2} i \\
    &= 4 + 3 + 2 + \sum_{i=0}^{1} i \\
    &= 4 + 3 + 2 + 1 + \sum_{i=0}^{0} i \\
    &= 4 + 3 + 2 + 1 + 0 \\
    &= 10
\end{align*}
\]

We could implement this as a function that calls itself until \( n \) becomes 0:

```c
int sum(int n)
{
    if (n == 0)
        return 0;
    else
        return n + sum(n-1);
}
```
Recursion is possible because information pertaining to each function call is saved on the program’s run-time stack. Recall that a stack is a data structure in which information is always added to and taken from the top. When a function is called, it is said to be activated and the following information pertaining to the function call is contained in its activation record:

1. The function’s local, automatically allocated variables.
2. The function’s parameters (which behave like local variables).
3. The return address to which control will return when the function has completed its execution.

This activation record is pushed onto the run-time stack. If that function calls itself (or another function), an activation record for the second function call is pushed onto the stack. In this way information pertaining to a number of successive function calls can be saved and referred back to. When the function call on the top of the stack is done, its activation record is popped off the stack and control returns to the function that called it and so on. What follows is a crude representation of the stack for the case in which a main function called the sum function with a value of 4.

```
function calls push activation records on stack

sum(4) return 4 + sum(3); 10
sum(3) return 3 + sum(2);  6
sum(2) return 2 + sum(1);  3
sum(1) return 1 + sum(0);  1
sum(0) return 0;          0
```

6.9 Recursion and Linked Lists

The following examples of recursive linked list processing involve calling the same function on successive nodes in the list until reaching a node where the base case is satisfied.\^2 At that point, control is returned back through each of the previous function calls until the head of the list is reached.

**Printing a list in order.** A simple way to understand this process is to think of the linked list as a passenger train in which each of the train cars represents a node in the list. Suppose the conductor of the train starts at the first car and collects tickets from the passengers in that car, moves to the next car and does the same, and so on until the last car is reached. At that point, he must turn around and return back through all of the cars to reach the first car. Since he collected the tickets on his way to the end of the train, he has nothing to do on his way back to the first car. This is an

\^2Another way to use recursion with linked lists is for a function to call itself with smaller and smaller versions of the list. This approach can be used for linked list sorting algorithms which will not be considered here.
example of tail recursion\(^3\), because the conductor does nothing on his way back but return through the cars. This example is analogous to using recursion to print a list in order. In pseudo-code, the function would look like this:

\[
\text{Print in order}
\]
\[
\text{If we're beyond the last node} \\
\text{return} \\
\text{Else} \\
\text{print this node's data item} \\
\text{call printInOrder on the next node} \\
\text{return}
\]

The actual code for the List class `printInOrder()` function requires a helper function that can be called recursively:

```cpp
void
printInOrderHelper(Node* ptr) 
{
  if(!ptr)
    return;
  else
    cout << ptr->myInfo << endl;
    printInOrderHelper(ptr->myLink); 
    return;
}
void
List::printInOrder() 
{
  printInOrderHelper(myHead);
}
```

Notice that the helper function is not a List class function and that its definition appears above the `List::printInOrder()` function.

Printin**g a list in reverse.** As our second example, suppose the conductor proceeds all the way from the first car to the last car, turns around, and collects the tickets on his way back. This would not be an example of tail recursion because he is indeed doing something on his return back through the cars. This example is analogous to printing the list in reverse order. We want to proceed all the way to the end of the list and then print each node's data item on the way back. In pseudo-code, the function would look like this:

\[
\text{printInReverse}
\]

\(^3\)In tail recursion, the only part of the activation record used after the recursive call is the return address. The compiler may therefore implement the repetitive process more efficiently as a loop rather than recursive function calls.
6.9. RECURSION AND LINKED LISTS

If we’re beyond the last node
return
Else
call printInReverse on the next node
print this node’s data item
return

The code for the printInReverse functions is very similar to the printInOrder functions. The only real difference is that the output statement and the function call in the helper function have been swapped:

void
printInReverseHelper(Node* ptr)
{
    if (!ptr)
        return;
    else
        printInReverseHelper(ptr->myLink);
    cout << ptr->myInfo << endl;
    return;
}

void
List::printInReverse()
{
    printInReverseHelper(myHead);
}

Finding an item in a list. As our third example, suppose you are in the first car and want to go to the snack car to buy a candy bar. You would start at the first car, proceed to the second car, and so on until you reached the snack car. At that point, you would buy a candy bar and return back through the cars until you reached the first car. Again, this would be an example of tail recursion because you would have nothing additional to do as you returned back through the cars. This example is analogous to looking through the list until we find a particular value. In this case, as soon as we find the value, we can return a value of “true” without having to go all the way to the end of the list. However, if we do go to the end of the list without finding the value, we will return a value of “false”.

In pseudo-code, this looks like

    find
    If we’re beyond the end of the list
        return false
    If this node contains the value we’re looking for
        return true
    Else
        return the result of calling find on the next node
Here is the code for the `find` function and its recursive helper function:

```c
bool findHelper(int value, Node* ptr)
{
    if(!ptr)
        return false;
    if(ptr->myInfo == value)
        return true;
    else
        return findHelper(value, ptr->myLink);
}

bool List::find(int value)
{
    return findHelper(value, myHead);
}
```

Elegant recursive programming with linked lists is the mainstay of languages such as Lisp and Scheme (a Lisp dialect — see [SF91]). Both are used extensively in artificial-intelligence research.

### 6.10 Sorting Linked Lists

The purpose of this section is to demonstrate a more sophisticated use of recursion on linked lists. A side benefit is that you will

- Understand the algorithms for Insertion, Selection, Merge-, and Quicksort
- Understand iterative implementations for insertion and selection sort (using simple arrays).
- Understand recursive implementation for insertion, selection, merge-, and quicksort (using linked lists).
- Be acquainted with the most efficient sorting algorithms for most situations.
- Be exposed to *nonlinear* recursion, recursion in which a problem instance is reduced to more than one smaller instance.

Web sites related to sorting algorithms (suggested by Rick Klevans, former CSC 311 instructor):

- [http://www.cs.rockport.edu/cs/javasort.html](http://www.cs.rockport.edu/cs/javasort.html)
- [http://www.ifi.uio.no/~larsga/download/SortDemo/](http://www.ifi.uio.no/~larsga/download/SortDemo/)

More information about sorting (and other) algorithms is available in [CLR90].
6.10. **SORTING LINKED LISTS**

6.10.1 Problem decomposition: a new way to think about recursion

Lists lend themselves easily to recursive programming because they have a simple recursive definition: a list is either empty or it is an item followed by a smaller list. The recursion is most evident if you think of a node pointer (Node *) as (being the way to get at the items in) a list. The List class is really only an empty shell that holds a pointer to the first node. So a node pointer is either null (empty list) or it points to a node containing an item and another node pointer (way to get at the rest of the items). This suggests that a natural way to decompose a problem about lists is to consider how to incorporate the first item into a solution for the remainder of the list.

A simple example is the reversal of a list. If the list is empty, its reversal is also empty. Otherwise you can obtain its reversal by adding the first item to the rear of the reversal of the remainder of the list. You might try to imagine that the “reversal of the remainder” takes care of itself by magic. Then you can take advantage of the work that recursion automatically does for you and concentrate on solving the problem given that you have already solved a smaller instance of it. To illustrate, suppose you want to reverse the list [1 2 3 4]:

Because you can rely on recursion you only need to concern yourself with how to combine the reversal of [2 3 4] with other data to get the result you want. The first item needs to be added to the rear of the solution returned by the recursive call. In C++ this works out to be (you can write a definition of added_to_rear soon):

```c++
Node * reversal(Node * L )
// return a new list that has the items of L in reverse order
{
    if ( !L ) return 0;
    else return added_to_rear( L->myInfo, reversal( L->myLink ) );
}
```

In terms of activation records, the reversal of [1 2 3 4] would look like:
reversal([]) return 0; []
reversal([4]) return added_to_rear(4, reversal([])); [4]
reversal([3 4]) return added_to_rear(3, reversal([4])); [4 3]
reversal([2 3 4]) return added_to_rear(2, reversal([3 4])); [4 3 2]
reversal([1 2 3 4]) return added_to_rear(1, reversal([2 3 4])); [4 3 2 1]

main()

added_to_rear can also be defined recursively. A new item added to the rear of an empty list is a list containing that item. Otherwise assume, by the magic of recursion, that the rest of the list has had the item added to the rear. Then all you need to do is link to the result.

```c
Node * added_to_rear( int item, Node * L )
{
    if ( !L ) return new Node( item );
    else {
        L->myLink = added_to_rear( item, L->myLink );
        return L;
    }
}
```

The node pointed to by L is already new (from a previous call to added_to_rear) so you simply reuse and return it instead of allocating another new node.

This method of reversal turns out to be inefficient for long lists, not because it is recursive (the overhead for maintaining activation records of recursive functions is minimal), but because the total number of calls to added_to_rear for reversing a list of n items is proportional to n² (trace it and see). There is an iterative reversal function that needs no helper and runs in time proportional to the length of the list — it repeatedly adds each item of the list to the front of the result:

```c
Node * reversal( Node * L )
{
    Node * retval = 0;
    while ( L ) {
        retval = new Node( L->myInfo, retval );
        L = L->myLink;
    }
    return retval;
}
```

Lest you get the impression that iterative solutions are sometimes superior to recursive ones, we illustrate, using this simple example, how every loop can be converted to a recursive helper function (this is not a proof, only an illustration). The helper function implementing the loop needs to have
6.10. SORTING LINKED LISTS

one argument for every variable that changes value within the loop. It needs to “return” any variables that are needed after the loop, either using the return mechanism or by having argument(s) for that purpose. In this case only retval is used after the loop. The helper function implements the body of the loop, in this case:

```c
Node * while_loop( Node * L, Node * retval )
{
    if (!L) return retval; // simulate loop exit
    else // simulate another loop iteration, giving variables their new values
        return while_loop( L->myLink, new Node( L->myInfo, retval ) );
}
```

```c
Node * reversal( Node * L )
{
    return while_loop( L, 0 ); // initialize loop variables
}
```

Looking again at activation records,

```
while_loop([], [4 3 2 1])   return retval; [4 3 2 1]
while_loop([4], [3 2 1])   return while_loop([], [4 3 2 1]); [4 3 2 1]
while_loop([3 4], [2 1])   return while_loop([4], [3 2 1]); [4 3 2 1]
while_loop([2 3 4], [1])   return while_loop([3 4], [2 1]); [4 3 2 1]
while_loop([1 2 3 4])   return while_loop([2 3 4], [1]); [4 3 2 1]
reversal([1 2 3 4])   return while_loop([1 2 3 4], []); [4 3 2 1]
```

This is tail recursion — in fact any loop can be simulated using tail recursion so that a programming language that allows recursion (as most do) has no need for loops! Most good compilers, especially if you specify optimization, detect tail recursion and generate code that leaves out the function calls (and storing of activation records). Thus the recursive implementation, when compiled, will be indistinguishable from the iterative one.

We make this point to emphasize the utility of thinking recursively. There are programming problems that have no simple and efficient iterative solution but that do have simple, elegant, and efficient recursive solutions. On the other hand, any problem that has a simple and efficient iterative solution has an (almost) equally simple and efficient recursive solution.

Getting back to sorting, the four algorithms described below illustrate two different ways to analyze a problem: decomposing the input and decomposing the output, and two different ways to split up a list: first item (node) versus rest of list and (roughly) in half. The first-rest splits lead to simple algorithms that require time proportional to \( n^2 \) on \( n \)-element lists (on average), but have other advantages: insertion sort is fastest when lists are almost sorted, selection sort (when adapted
to an array) is fastest when the time spent is dominated by data movement. The splits in half lead to algorithms whose time is proportional to \( n \log n \), which is faster than \( n^2 \) for long enough lists (usually 100-500 is long enough).

### 6.10.2 Insertion Sort

For the insertion sort, the list is decomposed into first and rest. You can apply the same kind of logic as with the first example of reversal. What operation needs to be done to modify the sorted list returned by a recursive call? In this case the first item needs to be inserted into the sorted rest of the list:

\[
\begin{array}{c}
\text{sort} \\
[3 \ 2 \ 1 \ 4] \\
\downarrow \\
\text{insert} \\
[1 \ 2 \ 3 \ 4]
\end{array}
\]

In all the sorting functions it is easier to reuse existing nodes instead of allocating new ones (in Section 6.11 we call this a destructive implementation). Thus, instead of inserting the first item, you insert the node containing it. The function to do this (also recursive) is presented later.

Using the train analogy to understand the whole process, the conductor moves forward through the train stopping in each car to put the first node of the current list somewhere in the car. On the way back, the conductor retrieves these first nodes and relinks each in the proper position among those already collected. The C++ code is in Figure 6.16.

You can analyze the recursive insertion as follows: the easy cases are (a) inserting into an empty list and (b) inserting at the front of a list; in other cases, the problem becomes one of inserting the item into a smaller list, one that includes all but the first element of the original list — see Figure 6.17. With debugging printout of both functions that is indented to show the depth of the run-time stack, the full execution of insertion sort on [3 2 1 4] is illustrated in Figure 6.18.

### 6.10.3 Selection Sort

**Algorithm Description.**

Suppose that instead of doing a first-rest split of the input as in insertion sort, you analyzed the output in terms of a first-rest split instead. What do you need to do to come up with the first node of the sorted list? How do you obtain the rest of the sorted list? Look at Figure 6.19. The first node contains the smallest item in the list and the rest of the list needs to be a sorted version of all but the smallest item.

The conductor in the train analogy would deposit the (node containing the) smallest item of the current list in each car while moving forward. On the way back all the conductor needs to do is
6.10. **SORTING LINKED LISTS**

Node * sortHelper( Node * L ) {
    Node * retval = 0;
    if ( L == 0 ) retval = 0;
    else {
        Node * restOfL = L->myLink;
        L->myLink = 0;  // unlink the first node
        retval = insert( L, sortHelper( restOfL ) );
    }
    return retval;
}

void List::sort() {
    myHead = sortHelper( myHead );
}

Node * insert( Node * to_be_inserted, Node * L ) {
    // PRE: to_be_inserted != 0 & & to_be_inserted->myLink == 0 & & L is sorted
    //      in ascending order
    // POST: retval == the nodes of to_be_inserted and L rearranged so that
    //          the myInfo members are in ascending order
    Node * retval = 0;
    if ( !L )
        retval = to_be_inserted;
    else if ( to_be_inserted->myInfo <= L->myInfo ) {
        // insert before first item
        to_be_inserted->myLink = L;
        retval = to_be_inserted;
    } else {
        // insert in rest of list
        L->myLink = insert( to_be_inserted, L->myLink );
        retval = L;
    }
    return retval;
}

Figure 6.16: The recursive sort function for insertion sort.

Figure 6.17: The recursive insert function for insertion sort.
-> sortHelper, L = [3, 2, 1, 4]
  -> sortHelper, L = [2, 1, 4]
  -> sortHelper, L = [1, 4]
    -> sortHelper, L = [4]
      <- sortHelper
        -> insert, to_be_inserted = [4], L = []
          retval = [4]
        <- insert
        retval = [4]
  <- sortHelper
  -> insert, to_be_inserted = [1], L = [4]
    retval = [1, 4]
    <- insert
    retval = [1, 2, 4]
  <- sortHelper
  -> insert, to_be_inserted = [3], L = [1, 2, 4]
    -> insert, to_be_inserted = [3], L = [2, 4]
      retval = [3, 4]
    <- insert
    retval = [2, 3, 4]
  <- insert
  retval = [1, 2, 3, 4]
  <- insert
  retval = [1, 2, 3, 4]
  <- sortHelper
Sorted version of the list is:
[1, 2, 3, 4]

Figure 6.18: Trace of insertion sort on [3 2 1 4].
6.10. Sorting Linked Lists

```
[3 2 1 4] [3 2 1 4]
   |           all but smallest
?   ?
  / \
[1]  [2 3 4]  [1]  [2 3 4]
 first  rest  first  rest
[1 2 3 4] [1 2 3 4]
```

Figure 6.19: How do you get the two pieces (first and rest) of a sorted list?

Collect the item in each car and add it to the front of the list. The C++ code for recursive selection sort is shown in Figure 6.20.

The decomposition of the list into smallest and all but smallest is accomplished by the recursive select function. Using the train analogy on the select function, the conductor does nothing while moving forward except for leaving the first item of the current list in each car. On the way back, the conductor holds the smallest item in the left hand and the remaining items in the right hand. When entering each car, the conductor examines the item left there: if the item in the left hand is smaller, the item from the car is put in the right hand; otherwise, the item in the left hand is put in the right hand and the car’s item is put into the left. Putting both the sortHelper and the select function together leads to behavior that would have been hard to invent from scratch. You should pause and reflect on the simplicity of writing one recursive function at a time and doing so by looking only at what happens during one recursive call (recursive calls on smaller instances happen by magic).

Selection sort can be adapted to an array (recall how insertion sort was done on an array in Chapter 1). Think in terms of placing the smallest item into the first (index 0) position of the array (by swapping it with the item already there) and then recursively sorting the sub-array that begins at the next position — see Figure 6.21. Pointer arithmetic is used here (A + 1 is the address of the beginning of an array containing all but the first item of A). Because the swap puts the smallest item into its final resting place, there is nothing left to do after the recursive call — this is tail recursion. An equivalent version of sort with a loop replacing the recursion is:

```cpp
void sort( int * A, int length )
{
    while ( length > 1 ) {
        int index = smallest_index( A, length );
        swap( A[ 0 ], A[ index ] );
        ++A; --length;
    }
}
```
void select( Node * L, Node * & smallest, Node * & all_but_smallest )
// PRE: L points to a list with at least one element (L != 0)
// POST: smallest == (a pointer to) a single-node list containing the
//       smallest item of (the list pointed to by) L
//       && all_but_smallest == a list of the remaining items
{
  if ( L->myLink == 0 ) {
    smallest = L;
    all_but_smallest = 0;
  }
  else {
    select( L->myLink, smallest, all_but_smallest );
    if ( smallest->myInfo <= L->myInfo ) {
      // L's first node gets added to the front of all_but_smallest
      L->myLink = all_but_smallest;
      all_but_smallest = L;
    }
    else {
      // L's first node has the smallest item:
      // (1) smallest gets added to the front of all_but_smallest
      smallest->myLink = all_but_smallest;
      all_but_smallest = smallest;
      // (2) L's first node becomes the new smallest
      L->myLink = 0;
      smallest = L;
    }
  }
}

Node * sortHelper( Node * L ) {
  Node * retval = 0;
  if ( L == 0 ) retval = 0;
  else {
    Node * smallest;
    Node * all_but_smallest;
    select( L, smallest, all_but_smallest );
    all_but_smallest = sortHelper( all_but_smallest );
    smallest->myLink = all_but_smallest;
    retval = smallest;
  }
  return retval;
}

void List::sort()
{
  myHead = sortHelper( myHead );
}

Figure 6.20: The recursive functions for selection sort.
void swap( int & x, int & y ) { int tmp = x; x = y; y = tmp; }

int smallest_index( int * A, int length )
{
    int min_item = A[ 0 ];
    int min_index = 0;
    for ( int i = 1; i < length; ++i ) {
        if ( A[ i ] < min_item ) {
            min_item = A[ i ];
            min_index = i;
        }
    }
    return min_index;
}

void sort( int * A, int length )
{
    if ( length <= 1 ) return;
    else {
        int index = smallest_index( A, length );
        swap( A[ 0 ], A[ index ] );
        sort( A + 1, length - 1 );
    }
}

Figure 6.21: Recursive selection sort on an array.
6.10.4 Merge-sort

Merge-sort introduces a new idea: let's split the input in half and sort each half recursively. What operation do we need to do in order to bring the two sorted halves together into a sorted whole (see Figure 6.22)? Call that operation merge and see if you can do it more efficiently than appending the two halves and sorting. Using the idea that led to selection sort, you decide the first node of the result must contain the smallest item. But now it's easier to find — it has to be the first item of one of the two lists.

![Merge-sort diagram](image)

This takes care of the hard case. When one of the lists is empty, the result is the other list. The code for merge is shown (along with merge-sort) in Figure 6.23. Figure 6.24 shows a full trace of merge on the example.

Merge-sort is an example of recursion that goes beyond simple iteration — writing a loop to accomplish the same thing requires complex bookkeeping. The train conductor analogy no longer works. Look at the structure of the recursive calls in a larger example, as shown in Figure 6.25. The progress of activation records is obtained as follows: trace the leftmost path downward, reversing the direction of the arrows as you go and pushing the activation for each recursive call you encounter on the stack. When you encounter a call that has no downward arrows, pop its record off the stack and follow the upward arrow from it. The actual trace of recursive calls is shown in Figure 6.26.

---

4The function as written in Figure 6.23, when compiled under g++, puts the recursive call for the second half first, so the trace in that case corresponds to tracing a rightmost path downward.
Node * merge(Node * L1, Node * L2) {
    // PRE: L1 and L2 are sorted in ascending order
    // POST: retval == the nodes of L1 and L2 rearranged so that the myInfo
    //       members are in ascending order
    Node * retval = 0;
    if ( L1 == 0 ) retval = L2;
    else if ( L2 == 0 ) retval = L1;
    else if ( L1->myInfo <= L2->myInfo ) {
        L1->myLink = merge(L1->myLink, L2);
        retval = L1;
    }
    else {
        L2->myLink = merge(L1, L2->myLink);
        retval = L2;
    }
    return retval;
}

Node * cut_in_half( Node * slow, Node * fast )
// cuts the list beginning at slow after the length(fast)/2-th node and
// returns a pointer to the second part (note: if slow == fast, the list is
// cut in half)
// Assumption: neither slow nor fast is empty, and slow is at least as long
// as fast
{
    Node * retval = 0;
    if ( ! fast->myLink || ! fast->myLink->myLink ) {
        retval = slow->myLink;
        slow->myLink = 0;
    } else {
        retval = cut_in_half( slow->myLink, fast->myLink->myLink );
    }
    return retval;
}

Node * mergeSort( Node * L ) {
    Node * retval = 0;
    if ( L == 0 || L->myLink == 0 ) retval = L;
    else {
        Node * secondHalf = cut_in_half( L, L );
        retval = merge( mergeSort( L ), mergeSort( secondHalf ) );
    }
    return retval;
}

void
List::sort()
{
    myHead = mergeSort( myHead );
}

Figure 6.23: The recursive functions for mergesort.
→ merge, L1 = [2, 3], L2 = [1, 4]
→ merge, L1 = [2, 3], L2 = [4]
→ merge, L1 = [3], L2 = [4]
→ merge, L1 = [], L2 = [4]
  retval = [4]
← merge
  retval = [3, 4]
← merge
  retval = [2, 3, 4]
← merge
  retval = [1, 2, 3, 4]
← merge

Figure 6.24: A trace of merge on lists [2 3] and [1 4].

merge

merge

merge

merge

merge

merge

[1 4 6 7]

[1 2 3 4 5 6 7]

Figure 6.25: The structure of recursion for merge-sort.
–> mergeSort, L = [6, 4, 1, 7, 2, 5, 3]
–> mergeSort, L = [6, 4, 1, 7]
–> mergeSort, L = [6, 4]
–> mergeSort, L = [6]
<– mergeSort, retval = [6]
<– mergeSort, retval = [4]
–> merge, L1 = [6], L2 = [4]
<– merge, retval = [4, 6]
<– mergeSort, retval = [4, 6]
–> mergeSort, L = [1, 7]
–> mergeSort, L = [1]
<– mergeSort, retval = [1]
–> mergeSort, L = [7]
<– mergeSort, retval = [7]
–> merge, L1 = [1], L2 = [7]
<– merge, retval = [1, 7]
<– mergeSort, retval = [1, 7]
–> merge, L1 = [4, 6], L2 = [1, 7]
<– merge, retval = [1, 4, 6, 7]
<– mergeSort, retval = [1, 4, 6, 7]
–> mergeSort, L = [2, 5, 3]
–> mergeSort, L = [2, 5]
–> mergeSort, L = [2]
<– mergeSort, retval = [2]
–> mergeSort, L = [5]
<– mergeSort, retval = [5]
–> merge, L1 = [2], L2 = [5]
<– merge, retval = [2, 5]
<– mergeSort, retval = [2, 5]
–> mergeSort, L = [3]
<– mergeSort, retval = [3]
–> merge, L1 = [2, 5], L2 = [3]
<– merge, retval = [2, 3, 5]
<– mergeSort, retval = [2, 3, 5]
–> merge, L1 = [1, 4, 6, 7], L2 = [2, 3, 5]
<– merge, retval = [1, 2, 3, 4, 5, 6, 7]
<– mergeSort, retval = [1, 2, 3, 4, 5, 6, 7]
Sorted version of the list is:
[1, 2, 3, 4, 5, 6, 7]

Figure 6.26: A trace of mergeSort on a larger list.
6.10.5 Quicksort

Quicksort is what you might come up with if you split the result of a sort in half and ask how the two pieces are obtained. Since that's too difficult, you aim for the next best thing: two sublists that follow one another in sorted order and are each half the size of the original "on average". Start by selecting a pivot element of the list. The list elements are considered relative to the pivot. Elements less than or equal to the pivot are put in one sublist and elements greater than the pivot put in the other. At this point all you have to do is sort each sublist recursively and append them, putting the pivot in between.

There are several ways to pick a pivot — among them are

- pick the first element — easy, but bad if list is sorted or almost sorted
- choose randomly — efficient on average
- median of the first, middle, and last elements — usually works well, but can be bad if input has an "organ pipe" shape (increasing then decreasing)

Quicksort for linked lists is illustrated in Figure 6.27. There are many efficient implementations for arrays, among them the one that is used by the Unix sort utility. Quicksort (for arrays) was originally invented by Hoare [Ho62]. A later paper by Sedgewick [Sed78] describes several variations and how they effect efficiency.

6.11 Linked-List Programming Exercises

6.11.1 Purpose

The purpose of these exercises is to build your mental muscles for programming with specific sets of rules while using your creativity to the utmost. Think all the way through each of them before consulting the solutions and don't be surprised if your solutions have nothing in common with the given ones (and yet still work!).

6.11.2 General Rules

The exercises described here all involve solving a problem that has inputs (usually one or more linked lists plus possibly some other parameters) and result(s) (again, usually one or more linked lists). Almost any problem involving linked lists can be subjected to the treatment suggested by these exercises. The inputs can be instances of a List class or pointers to nodes; same with the results. Inputs or results could be instances from which a member function is called, arguments to a function, or return values from a function, hence the use of the more general terms inputs and results. The computation (another general term) by which the problem is solved can involve any number of functions (member functions of List or Node classes or stand-alone functions).

The specific exercises described below call for implementations of class List member functions. There are two ways to classify each implementation.

1. Recursive/Iterative:
void partition(Node * p, int pivot, Node * & smaller, Node * & bigger)
    // creates two lists from the list starting at p, one containing
    // items smaller than (or equal to) the pivot, the other containing
    // items larger than the pivot
    
    if ( p == 0 ) {
        smaller = 0;
        bigger = 0;
    }
    else {
        partition( p->myLink, pivot, smaller, bigger );
        if ( p->myInfo <= pivot ) {
            p->myLink = smaller;
            smaller = p;
        }
        else {
            p->myLink = bigger;
            bigger = p;
        }
    }
    
Node * append(Node * L1, Node * L2)
    // POST: retval = a list of the nodes of L1 followed by those of L2
    
    Node * retval = 0;
    if ( L1 == 0 ) retval = L2;
    else {
        L1->myLink = append(L1->myLink, L2);
        retval = L1;
    }
    return retval;

Node * quickSort( Node * L ) {
    Node * retval = 0;
    if ( L == 0 || L->myLink == 0 ) retval = L;
    else {
        int pivotValue = L->myInfo;
        Node * pivotNode = L;
        Node * smaller;
        Node * bigger;
        partition( L->myLink, pivotValue, smaller, bigger );
        smaller = quickSort( smaller );
        bigger = quickSort( bigger );
        pivotNode->myLink = bigger;
        retval = append( smaller, pivotNode );
    }
    return retval;
}

void List::sort()
    { myHead = quickSort( myHead );
    }

Figure 6.27: The recursive functions for Quicksort.
CHAPTER 6. LINKED LISTS

- **Recursive** means that there are no loops of any kind in the computation — any repetition must be accomplished using recursion.

- **Iterative** means no recursive calls can occur in the computation — loops must be used for all repetition.

2. **Constructive/Destructive:**

- **Constructive** means that the inputs are left completely unchanged and exactly enough new nodes are allocated to produce the results.

- **Destructive** means that the results are constructed from nodes of the inputs. New nodes are allocated only if these don’t suffice. If the results have fewer nodes than the inputs, any excess nodes are deallocated. Naturally, the original inputs are destroyed (restructured) in the process.

These classifications yield four different implementations: constructive iterative, constructive recursive, destructive iterative, and destructive recursive. The goal of the exercises is to solve each problem in all four different ways.

**Special rule for constructive recursive implementations:** To keep constructive recursive implementations in the spirit of the kind of recursive thinking that makes the Lisp programming language so elegant, we add the following rule: *Constructive recursive implementations cannot reassign the link of a node after it has been initialized using the constructor.* This also rules out the trivial solution for some of these exercises: doing constructive recursive implementations by applying destructive recursive ones to copies of lists.

### 6.11.3 The Exercises

The above idea can be applied to almost any linked-list problem. Some examples are given below. Online files that simplify the process of doing these exercises and solutions to them are available online (details to be announced in class). There is a simple test program for each problem and a makefile to compile both the iterative and recursive versions of the test program (a single program can test both constructive and destructive implementations). The makefile assumes that the implementations are in four separate files named *bletch_co_itr.cpp* (constructive iterative), *bletch_co_rec.cpp* (constructive recursive), *bletch_de_itr.cpp* (destructive iterative), and *bletch_de_rec.cpp* (destructive recursive), where *bletch* is the name of the problem (function). All of these are member functions of class *List* as described earlier in the chapter (unless otherwise noted). The recursion is done by helper functions that take node pointer argument(s).

1. **all but last** — the result is a list with the same items as the instance except for the last item in the list. The function name is shortened to abl for file naming purposes.

2. **shuffle** (with a *List* argument) — the result is a list with the same items as the two lists (instance and argument). Items are taken alternately from the lists. For example, if the lists are [1 2 3] (instance) and [4 5 6 7] (argument), the result is [1 4 2 5 3 6 7]. Items from the longer list continue to be included when the shorter runs out. If either list is empty, the result is (a copy of) the other list.
6.12 Key Concepts

3. **splice** (with a List and an integer argument) — the result is a list with the same items as the two lists. The items from the argument list are inserted after the \( k \)-th node of the first, where \( k \) is the integer argument. For example, if the two input lists are \([1 \ 2 \ 3]\) and \([4 \ 5 \ 6 \ 7]\), and \( k = 2 \), the result will be \([1 \ 2 \ 4 \ 5 \ 6 \ 7 \ 3]\). If \( k \leq 0 \), the items of the argument appear before the instance in the result. If \( k \geq n \), the length of the instance, the items of the argument appear after the instance. If either list is empty, the result is (a copy of) the other list.

4. **removeAll** (member of class **OrderedList** with a **OrderedList** argument) — the result is a list containing exactly those items of the instance that do not appear in the argument. If the instance is empty, so is the result. If the argument is empty, the result is (a copy of) the instance. Hint: this can be done using a technique like the recursive **merge** function.

6.11.4 Suggestions

Every problem has its own quirks and the details must be analyzed with pencil and paper. There are lots of different ways to solve each problem - be creative! Try to make your solutions as simple, efficient, and elegant as possible.

I usually find the constructive recursive implementations easiest to deal with. The natural question to ask is: How is the solution constructed from the first item of the input and a solution to a smaller instance of the same problem (usually the same problem for the rest of the list or lists)?

From there, I proceed to the destructive recursive implementation by simply reusing the first node of a list rather than allocating a new one (and being careful to deallocate any node that is no longer needed).

Constructive iterative implementations, unless some sort of reversal is involved, usually require keeping track of the last node added to the solution because new nodes need to be added to the end of the list as the inputs are traversed. Without a pointer to the last node, every new node that is added requires a traversal of the whole list.

Finally, destructive iterative solutions require careful relinking of the existing nodes. The key is to ensure that at all times there is a way to get to (either by direct pointer or a chain of pointers) any node whose **myLink** field might yet have to be changed.

6.12 Key Concepts

**Node** - An aggregate data object containing a data item (the int **myInfo** field in most examples in this book) and a link (pointer) to the next node.

**Linked list** - A chain of nodes, the link of each node pointing to the next. A pointer to the first node defines the whole list since all other nodes are accessible from it.

**Traversal** - Visiting the nodes of a linked list in order from front to rear.

**Recursion** - A problem solving technique based on reducing an instance of a problem to a smaller instance of the same problem. A recursive function is a function that calls itself.

**Tail recursion** - Occurs when a recursive function returns immediately following a recursive function call without doing any additional processing.
**Runtime stack / Activation record** - The runtime stack contains activation records for each successive function call for an executing program beginning with the call to the `main` function. An activation record is pushed onto the stack when a function is called and popped off the stack when the function completes. An activation record contains the automatically allocated data for the function (sometimes referred to as local variables), the arguments passed to the function, and the address in the program’s code to which control should return (return address) when the function completes.
Chapter 7

Inheritance and Polymorphism

7.1 Inheritance

Inheritance involves the use of an existing class as the base for a new or derived class. The derived class inherits all of the member data and functions of the base class, but it can also contain additional data and/or function members. Inheritance is considered an “is-a” relationship. Each derived class object “is-a” base class object as well.

As a simple example of the inheritance relationship, consider a family named Smith with two children, Bob and Mary. The children “inherit” certain characteristics and functions of their family such as the same last name, the same street address, the same family friends, and participate in the same family activities. However, each child may also have a unique set of friends, possessions,
activities, etc. in addition to those that they “inherit” from their family. Bob “is-a” Smith and Mary “is-a” Smith, but a Smith is not necessarily a Bob or a Mary.

7.2 The Inheritance Hierarchy

A class which has been derived from a base class can in turn be used as the base class for another derived class. This feature can be used to create an inheritance hierarchy. Consider, for example, a library which makes use of a Book class. A Book can be further classified as a Reference Book or a Circulating Book. A Reference Book can be further classified as a North Carolina Reference Book. A Circulating Book can be a Fiction or Non-fiction Book. A Fiction Book may be a Mystery, Science-fiction, or Western. A diagram of this hierarchy appears in Figure 7.1. Notice that a Mystery “is-a” Fiction Book which “is-a” Circulating Book which “is-a” Book. This is an example of single inheritance because each class is derived from a single base class (which may or may not have been derived from a base class itself). There is also a phenomenon known as multiple inheritance in which classes are derived from more than one base class. Multiple inheritance could be used to derive a Mini-van class from a Car base class and a Truck base class since a mini-van contains features of both cars and trucks. Implementing multiple inheritance can be very complicated,\(^1\) so we will focus on single inheritance.

\(^1\)So complicated that a precise definition of the semantics (behavior) of C++ multiple inheritance still eludes programming-language experts. Other object-oriented languages such as Java have ruled out multiple inheritance.
7.2. THE INHERITANCE HIERARCHY

//: Book.cpp − Book class implementation file
//  Suzanne Balik, 28 Jul 1999

#include "Book.h"

Book::Book(const char* author, const char* title)
{
    myAuthor = new char[strlen(author) + 1];
    strcpy(myAuthor, author);
    myTitle = new char[strlen(title) + 1];
    strcpy(myTitle, title);
}

Book::Book(const Book& other)
{
    myAuthor = new char[strlen(other.myAuthor) + 1];
    strcpy(myAuthor, other.myAuthor);
    myTitle = new char[strlen(other.myTitle) + 1];
    strcpy(myTitle, other.myTitle);
}

Book::~Book()
{
    delete [] myAuthor;
    delete [] myTitle;
}

Book&
Book::operator=(const Book& rhs)
{
    if(this != &rhs) {
        delete [] myAuthor;
        delete [] myTitle;
        myAuthor = new char[strlen(rhs.myAuthor) + 1];
        strcpy(myAuthor, rhs.myAuthor);
        myTitle = new char[strlen(rhs.myTitle) + 1];
        strcpy(myTitle, rhs.myTitle);
    }
    return *this;
}

void
Book::print(ostream& out)
{
    out << setiosflags(ios::left) << setw(25) << myAuthor << setw(30) << myTitle;
}

Figure 7.3: Book class implementation file.
Figure 7.4: Reference Book class header file.

7.3 Inheritance Types/Member Access Specifiers

There are three different types of inheritance available in C++: public, protected, and private. Use of protected and private inheritance is rare and so we will consider only public inheritance.

There are also three different access specifiers for class members: public, protected, and private. We have already used the public and private access specifiers in the classes we have previously defined.

Clients using a class have access to the public class members, but not the private class members. If we use the public inheritance type to derive a class from a base class, the derived class will also have access to the public members of the base class, but not the private members. If we would like the derived class (and no one else) to have access to the base class members, we can designate them as protected. The use of public inheritance with public, protected, and private base class members can be summarized as follows:

1. public base class members can be accessed by anyone.
2. protected base class members can be directly accessed by base or derived class functions.
3. private base class members can only be directly accessed by base class functions.

7.4 Library Hierarchy Implementation

We will look at the implementation of several of the classes in our Library Hierarchy which appear in Figures 7.2 to 7.9.

The Book Class. The Book class serves as the base class for the hierarchy. In designing a base class, it is important to consider the data and functions that all of the derived classes have in
7.4. LIBRARY HIERARCHY IMPLEMENTATION

```cpp
//: RefBook.cpp - Reference Book implementation file
//  Suzanne Balik, 28 Jul 1999

#include "RefBook.h"

RefBook::RefBook(const char* author, const char* title, int refCode = 0)
    : Book(author, title), myRefCode(refCode) {}

RefBook::RefBook(const RefBook& other)
    : Book(other), myRefCode(other.myRefCode) {}

RefBook::~RefBook() {}

RefBook& RefBook::operator=(const RefBook& rhs) {
    if (this != &rhs) {
        delete [] myAuthor;
        delete [] myTitle;
        myAuthor = new char[strlen(rhs.myAuthor) + 1];
        strcpy(myAuthor, rhs.myAuthor);
        myTitle = new char[strlen(rhs.myTitle) + 1];
        strcpy(myTitle, rhs.myTitle);
        myRefCode = rhs.myRefCode;
    }
    return *this;
}

void RefBook::print(ostream& out) {
    Book::print(out);
    out << "Reference:" << myRefCode;
}
```

Figure 7.5: Reference Book class implementation file.

common. In this case, they all have an author and a title for data and need to be able to set the author and title as well as print the class information. We therefore decided to use the protected access specifier for our base class data:

```cpp
protected:
    char* myAuthor;
    char* myTitle;
```

This gives each of the derived class functions access to the author and title. We could have used private access instead, in which case only the Book class functions would have had direct access to the data.

The Derived Classes. We specify that the RefBook class is publicly derived from the Book class as follows:

```cpp
class RefBook : public Book {
```
Figure 7.6: North Carolina Reference Book class header file.

Notice that the NCRefBook class is in turn publicly derived from the RefBook class:

```cpp
class NCRefBook : public RefBook {

public:
    NCRefBook(const char* author = "", const char* title = "",
               int refCode = 0);
    NCRefBook(const NCRefBook& other);
    ~NCRefBook();
    NCRefBook& operator=(const NCRefBook& rhs);
    void print(ostream& out);

};
```

Constructors/Destructors. A derived class constructor can call the base class constructor in its initialization list:

```cpp
CircBook::CircBook(const char* author = "", const char* title = "")
    : Book(author, title), myCheckedOut(false) {} 
```

and a derived class copy constructor can call the base class copy constructor in its initialization list:

```cpp
CircBook::CircBook(const CircBook& other)
    : Book(other), myCheckedOut(other.myCheckedOut) {} 
```

The base class constructor is always called before the derived class constructor. In the opposite way, the derived class destructor is always called before the base class destructor. The Book class destructor makes use of the `virtual` keyword, which will be explained in Section 7.8:

```cpp
virtual ~Book();
```
7.4. **Library Hierarchy Implementation**

```cpp
:// NCRefBook.cpp - North Carolina Reference Book implementation file  
//  Suzanne Balik, 28 Jul 1999

#include "NCRefBook.h"

NCRefBook::NCRefBook(const char* author, const char* title,  
  int refCode = 0)  
  : RefBook(author, title, refCode) {}  

NCRefBook::NCRefBook(const NCRefBook& other) : RefBook(other) {}  

NCRefBook::~NCRefBook() {}  

NCRefBook& NCRefBook::operator=(const NCRefBook& rhs)  
{  
  if(this != &rhs) {  
    delete [] myAuthor;  
    delete [] myTitle;  
    myAuthor = new char[strlen(rhs.myAuthor) + 1];  
    strcpy(myAuthor, rhs.myAuthor);  
    myTitle = new char[strlen(rhs.myTitle) + 1];  
    strcpy(myTitle, rhs.myTitle);  
    myRefCode = rhs.myRefCode;  
  }  
  return *this;  
}

void NCRefBook::print(ostream& out)  
{  
  Book::print(out);  
  out << "NC Reference:" << myRefCode;  
}

Figure 7.7: North Carolina Reference Book class implementation file.

A base class destructor should always be declared virtual.

**Functions.** The derived classes inherit the Book class setAuthor and setTitle functions. Each of  
the derived classes overrides (redefines) the Book class print function. The CircBook class overrides  
the print function as follows:

```cpp
void CircBook::print(ostream& out)  
{  
  Book::print(out);  
  out << (myCheckedOut ? "Checked Out" : "Available");  
}
```

The Book class print function can be called by using the **scope-resolution operator** with the class  
name, i.e., Book::print(out); The CircBook class also defines two additional functions, checkOut
and checkIn.

A simple client program. Figure 7.10 shows a simple main program that illustrates the use of the classes. Output from the program is in Figure 7.11. The rules of inheritance allow an object of a derived class to be copied into an object of a base class one or more levels up the hierarchy. For example, the NCRefBook doesBook is copied into an ordinary book (two levels up) by the statement

\texttt{Book slicedBook3(doesBook);} 

Since a Book instance has no room for the extra data of a RefBook (an NCRefBook introduces no additional data), that extra data is not copied — see Figure 7.12. This is called \textit{slicing} — some of the data is sliced away in the act of copying.

The reverse, copying from a base class instance to a derived class instance, is not allowed — there is no obvious mechanism for initializing the additional data properly.

7.5 Polymorphism

\textit{Polymorphism} is defined as the quality or state of being able to assume different forms. It is implemented in C++ through the use of inheritance, virtual functions, and base class pointers to allow a function to behave \textit{polymorphically}. This means that the function appears to assume \textit{different forms} depending on the type of class object it is being used with. Before we look at an example of polymorphism, we will consider a situation where a function \textit{does not} behave polymorphically.

```cpp
#include
#define CIRC_BOOK_H
#ifndef
#include "Book.h"
#define CIRC_BOOK_H

class CircBook : public Book {
    public:
        CircBook(const char* author = ", const char* title = "");
        CircBook(const CircBook& other);
        CircBook();
        CircBook& operator=(const CircBook& rhs);
        void print(ostream& out);
        void checkOut();
        void checkIn();

    protected:
        bool myCheckedOut;
    };
#endif
```

Figure 7.8: Circulating Book class header file.
7.5. POLYMORPHISM

---

//: CircBook.cpp − Circulating Book class implementation file
// Suzanne Balik, 28 Jul 1999

#include "CircBook.h"

CircBook::CircBook(const char* author, const char* title)
    : Book(author, title), myCheckedOut(false) {}

CircBook::CircBook(const CircBook& other)
    : Book(other), myCheckedOut(other.myCheckedOut) {}

CircBook::~CircBook() {}

void CircBook::print(ostream& out)
{
    Book::print(out);
    out << (myCheckedOut ? "Checked Out" : "Available");
}

CircBook&
CircBook::operator=(const CircBook& rhs)
{
    if (this != &rhs) {
        delete [] myAuthor;
        delete [] myTitle;
        myAuthor = new char[strlen(rhs.myAuthor) + 1];
        strcpy(myAuthor,rhs.myAuthor);
        myTitle = new char[strlen(rhs.myTitle) + 1];
        strcpy(myTitle,rhs.myTitle);
        myCheckedOut = rhs.myCheckedOut;
    }
    return *this;
}

void CircBook::checkIn()
{
    myCheckedOut = false;
}

void CircBook::checkOut()
{
    myCheckedOut = true;
}

---

Figure 7.9: Circulating Book class implementation file.
//: main.cpp - main function to test Book hierarchy
// Suzanne Balik, 15 Jul 1999

#include <iostream.h>
#include "Book.h"
#include "RefBook.h"
#include "NCRefBook.h"
#include "CircBook.h"

int main()
{
    // create some Book instances
    Book donatedBook("Mark Twain", "Tom Sawyer");
    RefBook postBook("Emily Post", "Etiquette", 1045);
    NCRefBook doeBook("John Doe", "History of North Carolina", 3024);
    CircBook christieBook("Agatha Christie", "Murder on the Orient Express");

    // print information
    donatedBook.print(cout); cout << endl;
    postBook.print(cout); cout << endl;
    doeBook.print(cout); cout << endl;
    christieBook.print(cout); cout << endl;

    // see if checking in and out works on a CircBook
    christieBook.checkOut();
    christieBook.print(cout); cout << endl;
    christieBook.checkIn();
    christieBook.print(cout); cout << endl;

    // try "slicing" (converting to the base class means that data specific
    // to the derived class is lost)
    Book slicedBook1 = christieBook; // same as "Book slicedBook1(christieBook);"
    slicedBook1.print(cout); cout << endl;
    RefBook slicedBook2(doeBook); // no info lost, but incorrect identification
    slicedBook2.print(cout); cout << endl;
    Book slicedBook3(doeBook);
    slicedBook3.print(cout); cout << endl;

    // no slicing here, just a copy
    NCRefBook newNC = doeBook;
    newNC.print(cout); cout << endl;
}

Figure 7.10: A client program for the Book inheritance hierarchy.
7.6 Non-virtual functions

Suppose we wanted to create a list of books which contained a combination of the various class types. We could create an array of Book class pointers:

```c
const int MAX = 5;
Book* bookList[MAX];
```

and use it to store newly allocated instances of the various classes:

```c
bookList[0] = new Book("Mark Twain", "Tom Sawyer");
```

These assignments are legal — instances of RefBook, NCreferBook, and CircBook are also instances of book. In fact, because pointers are being assigned instead of objects, no slicing occurs: all of these pointers point to the beginning of the Book data. When there is additional data, it is contiguous with the book data and the compiler “knows” how to produce the code to access it. Suppose, in contrast to the execution of Book slicedBook3(doeBook);, illustrated in Figure 7.12, we had done the following:

```c
NCreferBook * doeBookPtr
    = new NCreferBook("John Doe", "History of North Carolina", 3024);
Book * bookPtr = doeBookPtr;
```

The result is shown in Figure 7.13. Both doeBookPtr and bookPtr store the address of the first data item in the object, myAuthor. Adding 4 bytes to that address gives the address of myTitle, adding 8 bytes gives myRefCode.

If we print our list of “books” contained in the bookList array:
Figure 7.12: Slicing when a derived class instance is copied to a base class instance.

```java
for (int i = 0; i < MAX; i++) {
    bookList[i]->print(cout);
    cout << endl;
}
```

we get the following output:

```
csc% booklist
Mark Twain          Tom Sawyer
Noah Webster        Webster’s Dictionary
John Doe            North Carolina Birds
Danielle Steel      Star
Jane Austen         Pride and Prejudice
```

This is not because data has been sliced away, but because a Book class pointer is being used to call the print function, so the compiler assumes that the Book::print function should be used. The compiler has no way of knowing when to call the appropriate derived class print function (the contents of the array might change while the program is running). The decision to use Book::print in all cases is called compile-time or static binding because the compiler binds the address of the Book::print function to the function call bookList[i]->print(); before execution. In some cases, this may be exactly what is desired.
Figure 7.13: No slicing occurs when pointers are copied.

```cpp
//: Book.h - Book class header file
// Suzanne Balik, 28 Jul 1999

#ifndef BOOK_H
#define BOOK_H

#include <string.h>
#include <fstream.h>
#include <iomanip.h>

class Book {

public:
    Book(const char* author = "", const char* title = "");
    Book(const Book& other);
    virtual ~Book();
    Book& operator=(const Book& rhs);
    void setAuthor(const char* author = "");
    void setTitle(const char* title = "");
    virtual void print(ostream& out);

protected:
    char* myAuthor;
    char* myTitle;
};

#endif
```

Figure 7.14: Book class header file with virtual print function.
7.7 Virtual functions

If we would rather have our list printed out using the appropriate derived class print functions rather than the Book class print function, we could use the virtual keyword when the Book class print function is first declared in Book.h (see Figure 7.14):

```cpp
virtual void print(ostream& out);
```

Any derived class print function will then be considered virtual as well, even if it is not explicitly declared virtual. Declaring a function as virtual instructs the compiler to implement run-time, dynamic binding instead of compile-time binding. The compiler implements special code so that at run-time the correct print function will be called based on the type of the object being pointed to. Recall that because we’re dealing with a Book pointer instead of a Book object, the data related to the derived class is accessible. A virtual function can only be expected to work correctly when it is called using a pointer or a reference to a base class object.

Now when the following code is executed:

```cpp
for (int i = 0; i < MAX; i++) {
  bookList[i]->print(cout);
  cout << endl;
}
```

the output looks like this:

```
csc% booklist
  Mark Twain  Tom Sawyer
  Noah Webster  Webster’s Dictionary  Reference:5867
  John Doe  North Carolina Birds  NC Reference:2456
  Danielle Steel  Star  Available
  Jane Austen  Pride and Prejudice  Available
```

In this case the print function appears to behave polymorphically, i.e., it acts differently depending on the type of object being pointed to. This is a very powerful object-oriented programming technique. If we decided to define additional derived classes each with their own version of the print function, the above code would call the appropriate print function without requiring any modification.

7.8 Virtual destructors

As mentioned previously, a base class destructor should always be declared as virtual. This is to insure that the derived class destructor(s) will be called as well as the base class destructor if a derived class object is deallocated using a base class pointer. For example,

```cpp
delete bookList[2];
```

would result in the destructors, ~NCRefBook(), ~RefBook() and ~Book(), being called in that order.
7.9. Abstract Base Classes

In our preceding example, it was possible (and made sense) to create a base class Book object. In our library example, donated or new books which had not been classified as circulating books, reference books, or whatever could simply be considered as Book class objects. However, sometimes it is desirable to create an inheritance hierarchy in which the base class is abstract. This means that it is not possible (and doesn’t make sense) to use the base class to create an instance of an object.

Consider a drawing program which displays various types of shapes on the screen – rectangles, triangles, squares, circles, pentagons, etc. While it is possible to write a function to draw a circle or a rectangle, it is impossible to create a function to draw a shape because a shape is a generic, undefined object. However, it would be very convenient for the drawing program to maintain a list of shapes and have the draw function polymorphically draw the shape correctly depending on its type. This is made possible in C++ by the use of an abstract base class which contains one or more pure virtual functions. A pure virtual function is a function that is declared but never defined in the abstract base class. It simply serves as an interface to the derived class functions that have the same signature (same name and argument list). In this example, the Shape class would be an abstract base class which contained a pure virtual draw function. Then classes such as Circle, Rectangle, Triangle, and so on could be derived from the Shape class. Each of these classes would define its own draw function. Classes which are derived from abstract base classes are also known as concrete classes.

The Shape Class. The header and implementation files for our Shape class appear in Figures 7.15 and 7.16. Rather than declare a pure virtual draw function, we have declared pure virtual area and type functions:

```cpp
virtual float area() = 0;
virtual const char* type() = 0;
```

Figure 7.15: Shape class header file.
//: Shape.cpp − Shape Abstract Base class implementation file
// Suzanne Balik, 23 Jul 1999

#include "Shape.h"
#include <string.h>

Shape::Shape(int number) : myNumber(number) {}

int
Shape::getNumber()
{
    return myNumber;
}

//: Rectangle.h − Rectangle Concrete Derived class
// Suzanne Balik, 23 Jul 1999

#ifndef RECTANGLE_H
#define RECTANGLE_H

#include "Shape.h"

class Rectangle : public Shape {

    public:
        Rectangle(int number, float width, float height);
        virtual ~Rectangle() {};
        float area();
        const char* type();

    private:
        float myWidth;
        float myHeight;
};

#endif

Figure 7.16: Shape class implementation file.

Figure 7.17: Rectangle class header file.
7.9. **ABSTRACT BASE CLASSES**

```cpp
//: Rectangle.cpp - Rectangle Concrete Derived class implementation file
//: Suzanne Balik, 23 Jul 1999

#include "Rectangle.h"

Rectangle::Rectangle(int number, float width, float height)
    : Shape(number), myWidth(width), myHeight(height) {}

float Rectangle::area()
{
    return myWidth * myHeight;
}

const char* Rectangle::type()
{
    return "Rectangle";
}
```

Figure 7.18: Rectangle class implementation file.

```cpp
//: Triangle.h - Triangle Concrete Derived class
//: Suzanne Balik, 23 Jul 1999

#ifndef TRIANGLE_H
#define TRIANGLE_H

#include "Shape.h"

class Triangle : public Shape {

public:
    Triangle(int number, float base, float height);
    virtual ~Triangle() {};
    float area();
    const char* type();

private:
    float myBase;
    float myHeight;
};
#endif
```

Figure 7.19: Triangle class header file.
Figure 7.20: Triangle class implementation file.

Figure 7.21: Circle class header file.
The = 0 at the end of the function declarations is what makes these virtual functions “pure”. This means that they cannot be defined in the Shape base class. A base class which has at least one pure virtual function is considered abstract and it therefore cannot be used to instantiate an object. The code:

```
Shape myShape(234);  //illegal!
```

would be illegal.

**The Derived Classes.** The code for the Rectangle, Triangle, and Circle derived classes appears in Figures 7.17 to 7.22. Notice that each of these classes simply declares and defines both the area and type functions.

**The shape Program.** A main function and a List class which maintains a list of shapes appears in Figures 7.23 to 7.25. The makefile to create the shape program as well as sample output appear in Figures 7.26 and 7.27.

### 7.10 Key Concepts

**Inheritance** - Using an existing class as the base for a new (derived) class. The derived class inherits all of the member data and functions of the base class, but it can also contain additional data and/or function members. Inheritance is considered an “is-a” relationship since each derived class object is a base class object as well.

**Slicing** - Occurs when a derived class object is assigned to a base class object. The base class object has no where to put any additional data contained in the derived class object so the extra data is said to be “sliced off”. 

---

```cpp
//: Circle.cpp - Circle Concrete Derived class implementation file
// Suzanne Balik, 23 Jul 1999

#include "Circle.h"

Circle::Circle(int number, float radius) : Shape(number), myRadius(radius) {}

float Circle::area()
{
    return 3.1416 * myRadius * myRadius;
}

const char* Circle::type()
{
    return "Circle";
}
```

Figure 7.22: Circle class implementation file.
:// main.cpp − main function for shape program
// Suzanne Balik, 23 Jul 1999
#include <iostream.h>
#include "Shape.h"
#include "Rectangle.h"
#include "Circle.h"
#include "Triangle.h"
#include "List.h"

int main()
{
    List myShapeList;

    myShapeList.insert(new Triangle (234, 12.0, 5.0));
    myShapeList.insert(new Circle (456, 2.5));
    myShapeList.insert(new Triangle (145, 1.0, 2.0));
    myShapeList.insert(new Rectangle (342, 12.0, 5.0));
    myShapeList.insert(new Triangle (598, 6.0, 2.0));
    myShapeList.insert(new Rectangle (632, 10.0, 5.0));
    myShapeList.insert(new Circle (212, 0.5));
    myShapeList.insert(new Circle (789, 9.3));

    myShapeList.print(cout);
}

Figure 7.23: shape program main function.

:// List.h − List class header file
// Suzanne Balik, 23 Jul 1999
#ifndef LIST_H
#define LIST_H
#include <iostream.h>
#include "Shape.h"

class Node;

class List {
    public:
        List();
        ~List();
        void insert(Shape* shape);
        void print(ostream& out);

    private:
        Node* myHead;
};
#endif

Figure 7.24: List class header file.
// List.cpp - List class implementation file
// Suzanne Balik, 23 Jul 1999

#include "List.h"
#include <iomanip.h>

class Node {
public:
    Node(Shape* shape, Node* next = 0) : myShape(shape), myLink(next) {}  
~Node() { delete myShape; }
    Shape* myShape;
    Node* myLink; 
};

List::List() : myHead(0) {}

List::~List() {
    Node* tmp;
    while (myHead) {
        tmp = myHead;
        myHead = myHead->myLink;
        delete tmp;
    }
}

void List::insert(Shape* shape) {
    myHead = new Node(shape, myHead);
}

void List::print(ostream& out) {
    Node* tmp = myHead;
    if (myHead) {
        out << "Number       Area       Type" 
        << "−−−−−−       −−−−       −−−−" << endl;
        out << setiosflags(ios::fixed) << setprecision(2);
        while (tmp) {
            out << tmp->myShape->getNumber()  // Calls base class getNumber function
            << "\t" << setw(10)
            << tmp->myShape->area()  // Calls derived class area function
            << "\t" << tmp->myShape->type()  // Calls derived class type function
            << endl;
            tmp = tmp->myLink;
        }
    }
}

Figure 7.25: List class implementation file.
#makefile for shape program
#Suzanne Balik, 15 Apr 1999

shape: main.o Shape.o Rectangle.o Circle.o Triangle.o List.o
g++ -o shape -g main.o Shape.o Rectangle.o Circle.o Triangle.o List.o

main.o: main.cpp Shape.h Rectangle.h Circle.h Triangle.h List.h
g++ -c main.cpp

Shape.o: Shape.cpp Shape.h
g++ -c -Wall -g Shape.cpp

Rectangle.o: Rectangle.cpp Rectangle.h Shape.h
g++ -c -Wall -g Rectangle.cpp

Circle.o: Circle.cpp Circle.h Shape.h
g++ -c -Wall -g Circle.cpp

Triangle.o: Triangle.cpp Triangle.h Shape.h
g++ -c -Wall -g Triangle.cpp

List.o: List.cpp List.h Shape.h
g++ -c -Wall -g List.cpp

---

<table>
<thead>
<tr>
<th>Number</th>
<th>Area</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>789</td>
<td>271.72</td>
<td>Circle</td>
</tr>
<tr>
<td>212</td>
<td>0.79</td>
<td>Circle</td>
</tr>
<tr>
<td>632</td>
<td>50.00</td>
<td>Rectangle</td>
</tr>
<tr>
<td>598</td>
<td>6.00</td>
<td>Triangle</td>
</tr>
<tr>
<td>342</td>
<td>60.00</td>
<td>Rectangle</td>
</tr>
<tr>
<td>145</td>
<td>1.00</td>
<td>Triangle</td>
</tr>
<tr>
<td>456</td>
<td>19.64</td>
<td>Circle</td>
</tr>
<tr>
<td>234</td>
<td>30.00</td>
<td>Triangle</td>
</tr>
</tbody>
</table>
7.10. **KEY CONCEPTS**

**Virtual function** - A base class function declared such that a particular definition of the function may be selected at run-time based on the class type of the object for which it is being called. The keyword, `virtual`, is used in the function declaration, e.g., `virtual void draw();`. A virtual function must be called using a pointer to an object for run-time (dynamic) binding to occur.

**Pure virtual function** - A virtual function which is declared in the base class but not defined. The declaration of the function is followed by `= 0` to designate it as pure, e.g., `virtual void draw() = 0;`.

**Polymorphism** - Making use of inheritance, virtual functions and base class pointers to derived class objects to allow a function to behave differently depending on the type of object for which it is being used.

**Abstract base class** - A base class which contains at least one pure virtual function. Since an abstract base class contains one or more undefined functions, it cannot be used to instantiate a class object. Classes derived from the abstract base class must provide a definition of any inherited pure virtual functions.
Appendix A

CSC 214 Rules

A.1 Academic Integrity in CSC 214

A.1.1 Frequently Asked Questions about Academic Integrity in Programming Courses

The actual academic integrity policy for CSC 214 appears later in this section and may also be accessed from the official CSC 214 web page.

1. Isn’t most programming done in teams? Why shouldn’t we be allowed to work in teams on programming assignments?

   In order to work effectively on a team you have to have well-developed individual skills, the integrity to take responsibility for your work, and the ability to recognize clear boundaries between your contributions and those of others.

2. But isn’t learning most effective in a team environment?

   Yes and no. The type of team learning that is most effective (and we strongly encourage this) is the study group, where a small group of people gets together regularly to go over lecture notes and brainstorm possible test questions based on those notes. By filling in details you may have missed and going over additional examples you solidify your understanding of the material. There is strong anecdotal evidence, however, that when programs are written as a team effort most of the work is done by one individual and only that individual learns the important lessons of programming. In a recent semester, 3 sections of CSC 210 had programming assignments done in teams of up to three people, while the other three sections emphasized individual work with strict enforcement of academic integrity. The department head did “exit interviews” of people from both groups after they had completed CSC 311. The people who had experienced team programming in CSC 210 all had more trouble with programming assignments in CSC 311 than their peers in the individual-work sections.
3. Why make such a big deal of academic integrity? Doesn’t everybody cut corners and cheat a little every now and then?

Do you want to be operated on by a surgeon who routinely had other medical students cover up his/her mistakes in med school? Or, more to the point, would you want to fly in a plane whose controller software was designed and implemented by a group of people who had never demonstrated the persistence, attention to detail, and ability to deal with negative feedback from compilers, linkers, etc., that it takes to design, implement, and debug a program on their own? Our exams can’t test for this, so a significant chunk of your grade needs to come from programming assignments, which we expect to be your own effort.

4. Isn’t it true that in most classes homework is designed to be practice or a way to learn the material that’s going to be on the test?

University policy (see www.ncsu.edu/ncsu/studaffairs/policies/code95.html) takes a different view. Think of homework as part of the final exam and the experience of doing it as essential to the learning that takes place in the course. If you want practice, do exercises in the textbook, get together with a study group and write small programs that exercise your understanding of particular concepts (we, the instructors and TAs, can give you ideas along these lines), but do not work on homework together.

5. When I use code from the text, lecture notes, or examples in the web, do I need to change the formatting or variable names to make it look different?

Most emphatically no! If you change variable names or format, it appears as if you did something wrong and were attempting to cover it up. When you borrow code from any source, the most important thing is to be up front about it by adding a comment to give appropriate credit to the source (this is the essence of academic integrity). Second, since CSC 214 teaches you about code reuse, you can demonstrate an important skill by using other code with only a minimal amount of change (or even whole cloth if possible). In a lot of settings, it is very important to make only essential changes to code, so that the diff utility (do man diff to find out more) gives an accurate impression of what needed to be fixed. Changes in format and variable names to “customize” the code to your own taste are frowned upon in most team-oriented projects. Also, the moss utility ignores such changes when determining whether two pieces of code are similar.

A.1.2 The Academic Integrity Policy for CSC 214

Programming assignments and exams in CSC 214 are designed to develop your individual programming skills, without which you cannot function effectively as a member of a programming team. Therefore all work in CSC 214, unless otherwise specified, is expected to be your own effort. Please read and sign the statement below, returning it to your section instructor at the beginning of the first class after the Homework 0 due date.
A.2. CSC 214 Documentation and Style Standards

Documentation and style standards are used by programmers to make their code readable. All your programs should conform to the following standards. If you submit code that deviates from the standards, expect some reduction in your grade for that assignment. Programs should be written so that they will compile without warnings using g++ (gcc) version 2.95.1 on Solaris 2.6 or version 2.91.66 on Linux. The version on Linux is the standard version for the EOS-Linux workstations in
the Leazar labs. On Solaris stations, however, you must add gcc295\(^1\). Source code for programs should be in files with a .cpp suffix (.h for header files).

### A.2.1 Documentation standards

Most of the programs you submit in CSC 214 will have multiple files containing code in C++ and other languages (e.g. makefile scripts). Every collection of submitted files should include (in the same directory as the other files) a text file called README that lists all the files in the directory and gives a brief description of each. The README file should also contain information about how the program(s) in the directory is (are) organized, how they should be compiled, and how to execute them.

To conform to conventional C++ programming style, delineate comments with //’s rather than with /*’s (comments in shell scripts and Makefiles begin with #). Each program file you submit should contain comments as follows.

1. A file header containing this information should be at the top of the file. (A sample header appears below.)

   (a) The top line of the file should be ///; followed by the file name, followed by a one-line description of the contents. This is sometimes referred to as a **slug line**. It facilitates getting a quick overview of all files in a multi-file program (by doing `fgrep //: *.*`). In place of slug lines, shell scripts generally begin with a comment that specifies the shell to be used (e.g. `#!/bin/csh`) and PostScript files begin with a specification of the PostScript version expected (e.g. `%!PS-Adobe-3.0`).

   (b) Personal information: your name, section, and instructor’s name.

   (c) History information: the date and time of the first successful compilation followed by dates, times, and brief descriptions of all bug fixes (give credit to anyone who helped in this part).

   (d) Overview: any useful information about the contents of the file that is not covered by comments on function headers.

An example of a C++ file:

```c
///: List.h - header for linked list operations
/// Matt Stallmann (mfms@eos.ncsu.edu), 30 Dec 1997, 11:54

/// MODIFICATIONS:
/// 30 Dec, 14:42 - fixed return type of insert() so that it returns
///        pointer to new list

/// OVERVIEW: All functions that modify a list return pointers to the
///        modified version of the list.
```

\(^1\)The default version, 2.8.1, has caused problems with programs relevant to this course, particularly those that interact with X11.
2. Major loops and conditionals that do not conform to any well-known idiom (use your judgment) should be commented. Comments before the code should be indented at the same level as the code. Comments within braces should be indented at the same level as the statements within. This simplified example demonstrates an appropriate commenting style (since this code is such a simple, well-understood idiom, it's not clear that comments are required at all).

```c
// Make sure that x holds the smaller of the two values.
if ( x > y ) {
    // Swap x and y.
    t = x; x = y; y = t;
}
```

A technically precise and concise way to do comments is to comment major loops with relevant loop invariants and other major blocks of code with assertions. Two simplified examples follow.

```c
if ( x > y ) {
    t = x; x = y; y = t;
} // ASSERT: x <= y
```

The comment says: at this point in the code, the programmer guarantees that x achieves the guaranteed condition and the programmer knows that the guarantee can be relied upon in the code that follows the comment.

```c
i = 0;
while ( i < LENGTH ) {
    A[i] = i;
    i++;
}
```

Here the comment says that the condition will be true before each test of the condition in the while statement, no matter how many times the loop is executed. The reader can easily verify that the comment is true before the loop is executed (it says nothing in that case), that it continues to be true after each additional execution of the loop body, and that, when the program exits the loop, A[0] == 0, A[1] == 1, ..., A[LENGTH-1] == LENGTH-1.

3. Functions must have their own headers. A header contains a description of what the function does. A header should tell the relationship between the returned value (if there is one) and the arguments. It should tell what side effects occur, including changes to global variables and actual arguments. This should be done with sentences that mention each argument as well as a listing of all the arguments along with descriptions of how each is used. Below is a stylistically correct function header.
void writeCheck (double debit, double& balance, bool& overDrawn);
// Arguments:
// debit - amount of money to be debited from the account
// balance - in - amount of money in the account
// - out - amount of money after recording the debit
// overdrawn - out - true if and only if balance is now less than 0
// Side effect: Decrease the balance to reflect a withdrawal of the debit.

The term “in” indicates the value of the argument when the function is entered. The term “out” indicates the value of the argument when the function is completed.

An alternate way to do function comments, recommended by many authors, is called contract programming. Each function is documented with logical pre- and post-conditions. A precondition is something that needs to be true in order for the function to behave as advertised. For example,

double squareRoot(double x);
// PRE: x >= 0

(this function will not compute a correct result if x < 0). A post-condition is something that will be true when the function is done with its execution. For example,

void writeCheck (double debit, double& balance, bool& overDrawn);
// POST: balance == balance<in> - debit &
// overDrawn == true if and only if balance < 0

Here, balance refers to the output value of balance (the post-condition is true after function execution). The notation balance<in> is used to refer to the input value.

Typically, a function has both pre- and post-conditions. The squareRoot function could have had a comment that said

// POST: retval == the (positive) square root of x

The notation retval is used to denote the return value of a function that has one. The above comment is probably superfluous since it’s understood from the function name.

4. Every variable and constant identifier whose use in the program is not obvious from the name should be commented when it is declared. This includes both global identifiers and local identifiers in a block. After each declaration, place a comment telling the purpose of the identifier.
const taxRate = 0.045; // sales tax rate on consumer goods
const maxLength = 200; // maximum number of items in a bill

// NOTE: the following 3 comments are probably overkill
struct itemInfo { // maintains information on a single item
    String item; // name of the item
    double price; // price of the item
};
typedef itemInfo[100] bill; // type for list of items purchased

enum Days {Mon, Tue, Wed, Thu, Fri}; // days of the week

double owes; // total amount of bill tallied so far
bill customer; // list of items purchased
int i; // index for the i-th item purchased
days weekday; // a day of the week

If (a) a variable name is self-documenting, and (b) loop invariants are given for major loops that affect the value of the variable, there is no need to comment the declaration.

A.2.2 Style Guidelines

1. Always use descriptive variable names. For example, the identifier `userName` is more descriptive than `un`; `testAverage` is more descriptive than `t`. Variable and function names that are more than one word should either use a capital letter at the beginning of all but the first word, as in `numberOfBuckets`, or an underscore between words, as in `number_of_buckets` (one or the other - try to stick to whichever you choose consistently).

2. Avoid “magic numbers” - use named constants or `enum`'s instead (these should be in all capital letters to distinguish them from other variables). The only numbers that should appear in your programs (other than as initial values of constants) are 0, 1, and -1.
   Bad:
   
   ```c++
   for ( int i = 0; i < 100; i++ )
       A[i] = 2;
   ```

   Better:

   ```c++
   const int DATA_POINTS = 100;
   enum Acidity { ACID, BASE, NEUTRAL };
   ...
   for ( int i = 0; i < DATA_POINTS; i++ )
       A[i] = NEUTRAL;
   ```

3. C++ is case sensitive. Put most of your code in lowercase (except for constants and `enum` values). Put comments in mixed case.
4. Indentation in your program should indicate the structure of your code.

If a statement is a continuation of the line above, it should be indented at least 3 spaces more
than that line. For example,

```java
if (( !overDrawn ) && ( timeOfDeposit <= today )
    && ( loanRequest < maxLoan ) && ( loanRequest > 0 ))
    makeLoan( );
```

Statements that make up the body of a loop or selection should be indented two or three spaces.
Indentation and other style decisions within each file should be consistent (for example, every
if statement should be indented the same way and the curly braces of every inner block should
either all line up or all not line up). Here are some examples.

```java
sum = 0;
// INV: sum == the sum of all positive entries among A[0], ..., A[i-1]
for ( int i = 0; i < LENGTH; i++ ) {
    if ( A[i] > 0 )
        sum += A[i];
}
if ( grade < 0 )
    grade = 0;
else {
    posGrades += grade;
    numPos++;
}
// Solve problems until the user wants to quit.
do {
    showMenu( problem );
    if ( problem != quit )
        solve( problem );
} while ( problem != quit );
// How many values sum to less than max?
numbers = -1;
sum = 0;

// INV: sum == the sum of all inputs so far
// && numbers == one less than the number of inputs so far
while ( sum <= max ) {
    cin >> value;
    numbers++;
    sum += value;
}
```

cout << "There were " << numbers << " with sum less than ";
cout << max << "." << endl;
5. The braces for a function should be indented at the same level as that function definition.
   All functions should be aligned with the left-hand margin. The left brace (initial brace) for a
   conditional or a loop should be on the same line as the condition. The right brace (terminal
   brace) should be on a separate line, indented the same as the reserved word if, while, etc.
   Below is a correct example.

   ```
   void ascendingOrder ( double& x, double& y )
   {
       double temp; // used to swap values if necessary
       if ( x > y ) {
           temp = x; x = y; y = temp;
       }
   } // ascendingOrder
   ```

6. Do not write any function to extend over a page in length. (No function should contain more
   than 25 lines of code.)

7. Use global variables sparingly.

8. The header file for a class should be called `ClassName.h` and the implementation called
   `ClassName.cpp`. File names for client programs should begin with lower-case letters.

9. In a class declaration,

   (a) the `public:` section should precede the `private:` section,

   (b) the opening brace should be on the same line as the class name,

   (c) the labels `public:` and `private:` should be on the left margin (as should the keyword
       `class`),

   (d) member function and member data declarations should be indented two or three spaces,

   (e) data member names should start with the prefix `my`

   (f) the closing brace should be on a line by itself, on the left margin (remember the semicolon!)

   For example,
const int MAX_ELEMENT = 31;
class IntSet {
    // in comments below, S denotes the current set object
    // all element of S must be integers in the range 0-MAX_ELEMENT
    public:
    IntSet();
    // POST: S == emptyset

    void add(int item);
    // PRE: 0 <= item < MAX_ELEMENT
    // POST: S == S<in> U {item}

    void remove(int item);
    // PRE: item is an element of S
    // POST: S == S<in> - {item}

    bool isMember(int item) const;
    // POST: retval == true if and only if item is an element of S

    private:
    bool myArray[MAX_ELEMENT+1];
};

10. When defining class member functions, put the return type of the function on a separate line
    so that the class name appears in the leftmost column. For example,

    const char *
    Token::string() {
        // body of the member function string() of class Token
        ...
    }

    Most of these guidelines are arbitrary, but important for mutual readability of programs within
    a group. You are likely to be asked to follow different guidelines about indentation, commenting,
    etc. in every work environment you encounter. Heated debates about stylistic issues are a common
    phenomenon. See Marshall Cline’s document on Coding Standards for a more thorough discussion
    is also important to remember that style has nothing to do with syntactic correctness. The following
    is a syntactically correct program that will compile on any valid C++ compiler and print i=44 when
    it executes.

    int main()
    ){int i=44
    cout<<"i="
    <<i<<endl;}
Appendix B

Dealing with Errors

B.1 Common Errors in C++ Programming

B.1.1 General Strategies

Here are some suggestions in tracking down the meaning of error messages from the preprocessor (cpp), compiler (gcc), and loader (ld). For execution-time errors such as Segmentation fault or Bus error, see Section B.2 on using the gdb debugger.

- Look for and fix the very first syntax error before trying to do anything about the others. If you’ve left out a semicolon, parenthesis, or brace, or misdeclared a variable or function, the compiler is likely to misunderstand the rest of the program and issue a lot of nonsensical messages. Don’t be intimidated by several screens full of error messages — they often go away after you fix one simple mistake.

- Most of the time the line number given with an error message is the line on which the error actually occurs. Some errors, such as parse errors, however, are notorious for reflecting a mistake several lines earlier or even in an included file. Try putting ;} on the line right above where the message says the error was and seeing if the parse error (or some other error) now refers to that line. Keep moving the ;} earlier in the file until you locate the source of the error.

- The loader prints out mangled names when reporting undefined symbol or multiply-defined symbol names. To unmanagle the names and find out what’s really the source of the problem, do the following. Suppose, for example, that you get this error:

```
% g++ badlink.cpp fun1.cpp
Undefined first referenced
symbol in file
fun1_FR7istreamPci /var/tmp/cca002MC1.o
ld: fatal: Symbol referencing errors. No output written to a.out
```

Then you can do (using the c++filt program)

```
```
% add gnu
% echo fun1__FR7istreamPci | c++filt
fun1(istream &, char *, int)

and see which function is undefined. To further isolate the error, do

% fgrep -n 'fun1(' *.h *.cpp
badlink.cpp:3:int fun1(istream& in, char *p, int x);
badlink.cpp:8: return fun1(cin, "test", 3);
fun1.cpp:1:int fun1(char * s, int i)

and you can see that fun1 is declared and used in badlink.cpp in a way that is inconsistent
with its definition in fun1.cpp (the -n switch causes line numbers to be shown for all lines
containing the search string).

- The Unix utility, nm, can be used to print the symbol table (name list) for one or more object
  files:

csc% nm badlink.o fun1.o
badlink.o:

[Index]  Value  Size  Type  Bind  Other  Shndx  Name

[2] | | 0| 0|SECT |LOCL |0 |3 |
[5] | | 0| 0|SECT |LOCL |0 |2 |
[6] | | 0| 0|SECT |LOCL |0 |4 |
[7] | | 0| 0|NOTY |GLOB |0 |UNDEF |_Q_qtod
[4] | | 0| 0|NOTY |GLOB |0 |4 |__FRAME_BEGIN__
[9] | | 0| 0|NOTY |GLOB |0 |UNDEF |__throw
[1] | | 0| 0|FILE |LOCL |0 |ABS |badlink.cpp
[10] | | 0| 0|NOTY |GLOB |0 |UNDEF |cin
[8] | | 0| 0|NOTY |GLOB |0 |UNDEF |fun1__FR7istreamPci
[3] | | 0| 0|NOTY |LOCL |0 |2 |gcc2_compiled.
[11] | | 0| 64|FUNC |GLOB |0 |2 |main

fun1.o:

[Index]  Value  Size  Type  Bind  Other  Shndx  Name

[4] | | 0| 0|SECT |LOCL |0 |2 |
[5] | | 0| 0|SECT |LOCL |0 |3 |
[3] | | 0| 0|NOTY |LOCL |0 |3 |__FRAME_BEGIN__
[1] | | 0| 0|FILE |LOCL |0 |ABS |fun1.cpp
[6] | | 0| 48|FUNC |GLOB |0 |2 |fun1__FPci
[2] | | 0| 0|NOTY |LOCL |0 |2 |gcc2_compiled.
This tells us, for example, that cin and fun1_FR7istreamPci are undefined in badlink.o and that fun1_FPci is defined globally in fun1.o (all the function names are mangled, of course).

B.1.2 Actual Error Messages from the g++ Compiler

In its error messages, g++ does not distinguish between built-in types such as int and user-defined classes — the term “class” is a synonym for “type”. Also, there is no distinction between a function argument that is an array and one that is a pointer to the type of element in the array. For example, if you declare a function prototype sort( int A[], int length ), g++ refers to it as sort( int * A, int length ).

Bad magic number — This actually occurs either at load time or when you attempt to execute a program. The magic number is a code near the beginning of an object file or executable file that specifies what machine architecture the file can be executed on. At load time, a bad magic number means that one of the object (.o) files being linked is incompatible with the others. At the start of execution, it means the executable was compiled a linked for a different architecture than the one you’re currently running (e.g., you compiled on Linux and are trying to run the program on Solaris).

Base operand of ‘-‘ has non-pointer type — You are trying to use - when the left (base) operand is not a pointer. You probably should be using . (dot) instead.

Bus error — This occurs at execution time — see Section B.2 or Appendix E.

Name lookup of ... changed for new ANSI ‘for’ scoping — This occurs when you try to use a variable that was declared in a for-loop header outside the body of the loop, as in

```cpp
for ( int i = 0; i < 3; ++i ) cout << i << endl;
```

...is only defined in the loop body

ANSI (American National Standards Institute) C++ disallows such usage even though many early C++ compilers allowed and even encouraged it.

New declaration ... ambiguates ... old declaration — You have declared two functions whose only difference is the return type. C++ regards these as being the same function.

No match for ... — Something like ‘X& == X&’ might appear where the ...are. You’re trying to use a C++ operator (in this case ==) and one of the operands is an instance of a (user-defined) class (in this case both instances are of class X) for which the operator has not been overloaded.

No matching function for call to ... — You’ve called a function but the arguments you’ve given do not match any declared prototype. This error also occurs when you declare an instance of a class and your declaration does not match any of the available constructors. For example, if Book is a class, the declaration Book shelf[100] requires that a null constructor be present (this is true any time an array of class instances is declared).

Parse error before ... — The compiler has encountered something unexpected. If the error is near the beginning of a source file, the mistake might be at the end of an included header file. Common causes of parse errors are missing semicolons, missing ’s, use of a keyword or class name where a variable name is expected, and unbalanced parentheses.
Prototype for ... does not match any in class ... — You are trying to implement a member function of the given class and the prototype does not match any that have been declared for that class. This often occurs if you declare a member function as const in the header file and then leave out the const in the implementation or vice versa. C++ regards const as an important part of the prototype: for example, int value() and int value() const are two different (member) functions.

Redefinition of ‘class ...’ — A class has been defined twice. Most likely, you forgot to put #ifndef directives into the header file that declares the class.

Request for member ... in ..., which is of non-aggregate type ... — An aggregate type is a struct or class, that is, anything that can have members accessible via the . operator from an instance. You are probably trying to use . instead of -> to access a member from a pointer (pointers, built-in types, and arrays are non-aggregate).

Return type specification for constructor invalid — You have put the name or declaration of a type or class in front of the definition of a constructor (recall that constructors and destructors have no return types). This error sometimes occurs when you leave out the semicolon at the end of a class declaration and the first thing after it is the definition of the constructor. Be on the lookout for a missing semicolon in a header file if the constructor definition is the first thing in the file being compiled.

Segmentation fault — This occurs at execution time — see Section B.2 or Appendix E.

Suggest parentheses around assignment used as truth value — You probably used = instead of == to compare two values. This is only a warning because a statement like

\[
\text{if ( } x = y \text{ ) } \{ \ldots \}
\]

is perfectly legal in C++ (holdover from C) if \( x \) and \( y \) are integer variables (the assignment operator returns a value).

Syntax error before ... — Similar to parse error but usually less specific as to location. The compiler has gotten off track and the real error may be much earlier.

Undefined function — Usually refers to a variable rather than a function. Make sure the offending identifier is declared in the current scope. If it’s a data member of a class, make sure the current scope is that of a member function. You might have neglected to put the scope resolution, ClassName:: in front of the function name.

## B.2 Using the GDB Debugger

Suppose you run your program and end up with a segmentation fault. One possibility is to add a lot of debugging printout to trace the location of the error. Another possibility is to use the \texttt{gdb} debugger. Gdb will work only if you compiled your code using \texttt{g++} with the \texttt{-g} switch. You also need to do \texttt{add gnu} before running \texttt{gdb}.

You run \texttt{gdb} from the command line by saying \texttt{gdb name}, where \texttt{name} is the name of the executable program (e.g. \texttt{a.out} if no specific name was given when the program was linked). In response to the (gdb) prompt, type \texttt{run} followed by whatever command-line arguments (if any) you
want to specify. When a runtime error occurs, gdb will print a message telling you the file name
containing the statement whose execution caused the error and the line number of the statement.

This information may not be enough, but various commands in response to the prompt allow you
to poke around. The most useful command is where — it gives you (in reverse order) the sequence
of function calls that led to the error. Each call is annotated with file name and line number so you
can track what happened in your source code.

You can also look at values of variables with the command print data, where data is a simple
variable, array, structure/class, or any other reference to an object that’s legal in the current scope
(for example, name.myName[4]).

To change the current scope (initially that of the function or block in which the program crashed),
you can use the command up to get to function that called the current one or down to get back from
the most recent up. The up and down commands allow you to examine all the activation records on
the runtime stack (see Section 6.8).

The debugger can also be helpful in tracking down errors that don’t lead to segmentation faults.
Before issuing the run command you can instruct gdb to stop each time it calls a specific function
be setting a break point — type break function_name, where function_name is the name of the
function (give the name of a class member function using the scope resolution operator: for example,
break Book::print). When the program halts there you can examine values of variables as in the
case of a segmentation fault. Then you can use then cont command to resume execution.

Another possibility after reaching a breakpoint is to resume execution one statement at a time.
The commands next and step each execute one statement and then stop. The difference between
the two is that step, when it encounters a function call will execute that function one statement at
time, thus giving all the gory details. The next command, on the other hand, regards all function
calls within a statement as being part of the current statement and does not stop until the statement
is done. You don’t have to keep typing step or next when examining a lengthy execution, by the
way — hitting [Return] or [Enter] on your keyboard tells gdb to repeat the previous command.

Other information about gdb may be obtained by doing man gdb or by saying help in response
to the (gdb) prompt. Some editors, such as emacs, allow you to run gdb conveniently — in emacs,
for example, a window shows you the place in the file where the program stopped and you can set
breakpoints from the editor. A version of gdb with a graphical user interface, xxdgdb, is also available,
as are other window-based debuggers, such as ddd. You can spend a lot of time becoming an expert
at using a debugger, but most errors can be found quickly by examining the code, knowing your
own weaknesses (if it’s your own code you’re debugging), and simple uses of a debugger to locate
the source of the error.
Appendix C

The sxwin Window Library

The sxwin window library is designed to give easy access to the much more complicated X11 library on our system (for a description of the latter, see [Jon89]). The interface has a main part, sxwin.h, shown in Figure C.1, with a constructor, destructor, and private copying functions to prevent copying of sxwin data (since it would be confusing to have two data structures related to the same on-screen window).

There are four additional pieces that define most of the functionality:

1. sxwin_const.h (Figure C.2) defines of constants used by sxwin to communicate.

2. sxwin_input.h (Figure C.2) declares functions that allow your program to sense what is happening with keyboard, mouse, and screen. The detection of a WindowRefresh event allows you to redraw the window when necessary (when what once was hidden comes into view). Figure C.7 illustrates what the “event loop” in a typical windowing program might look like using sxwin.

3. sxwin_drawing.h (Figure C.4) declares functions that change the appearance of the window (graphics only, no text). All graphics are done in exclusive or mode — a pixel can be erased by drawing it a second time. A disadvantage of this is that there will be small white spots where lines of a drawing intersect,

4. sxwin_drawing.h (Figure C.5) declares functions that deal with text. There is no need to be concerned about the class String, used for all string arguments — a C-string will convert automatically to a String via a constructor. The diagram in Figure C.6 illustrates the meaning of baseline, font ascent, and font descent. The width of a string may be different in different fonts because the widths of letters may vary.

The sxwin window library implementation file, sxwin.cpp, must be compiled with the switch -I/usr/X11R6/include. Programs that use the sxwin window library must be linked with the object files sxwin.o and String.o and use the -lx11 switch. On some systems the -L/usr/X11R6/lib switch is also needed during linking.
//: sxwin.h - header for a simple X11 interface
// Matt Stallmann (ca. 1994, with additions and modifications since then)

#ifndef __SXWIN_H
#define __SXWIN_H

#include "String.h"
#include "sxwin_const.h"

class sxwin {
    // private data is hidden behind a generic pointer: this allows data
    // structure to be modified without changing this file (sxwin.h)
    struct HiddenWindowData;
    HiddenWindowData* data_
;
    // don’t allow sxwin’s to be copied!

    sxwin( const sxwin & );
    sxwin & operator = ( const sxwin & );

    public:
    sxwin( int x = 5, int y = 5, int width = 500, int height = 300,
            const String title = "" );
    // create a new window with the given width and height at
    // location (x,y) on the screen; title is the name on the title bar

    ~sxwin();
    // erase this window and clean up storage

    #include "sxwin_input.h"
    #include "sxwin_drawing.h"
    #include "sxwin_text.h"
};

#endif

Figure C.1: The sxwin class header file.
/*: sxwin_const.h - constants and enum types related to sxwin */

// character codes for non-printing characters available via getNextEvent
const char RETURN = 13;
const char BACKSPACE = 8;
const char DELETE = 127;
const char TAB = 9;
const char ESC = 27;

enum EventType {
    LeftPress, // left mouse button has been pressed
    LeftRelease, // left mouse button has been released
    MiddlePress, // middle mouse button has been pressed
    MiddleRelease, // middle mouse button has been released
    RightPress, // right mouse button has been pressed
    RightRelease, // right mouse button has been released
    MouseMotion, // the mouse has moved to location (x,y); generated only
    // if mouse motion tracking is turned on
    KeyStroke, // a key on the keyboard has been pressed (sxwin
    // ignores any key that is neither printable
    // ASCII nor one of the codes defined above
    WindowRefresh // a formerly hidden part of the window is now in view
};

enum TrackingState {
    OFF = 0,
    ON = 1
};

Figure C.2: Constants for the sxwin class.
Figure C.3: sxwin member functions that respond to input events or allow the programmer to gather information about a window.

Figure C.4: Drawing functions of the sxwin class.
//: sxwin_text.h − sxwin member functions related to text and fonts

void drawString(int x, int y, const String str);
// write the string of characters str starting at location with
// baseline height at y and left edge at x

void drawChar(int x, int y, char c);
// same as drawString, except only a single character is drawn

unsigned int fontAscent() const;
// returns the number of pixels that the tallest character extends above
// the baseline

unsigned int fontDescent() const;
// returns the number of pixels that the character with the longest "tail"
// extends below the baseline

unsigned int stringWidth(const String stg) const;
// returns the width of the given string in pixels

Figure C.5: Functions that manipulate text in sxwin windows.

Figure C.6: Measures of a font.
//: event_loop.cpp - program that prints information about X11 events on
//:                  the xterm from which it is run until user types a 'Q'

#include "sxwin.h"

int main() {  
sxwin mainWindow( 10, 10, 400, 400, "CSC 210 Window" );  
mainWindow.mouseTracking();  
bool done = false;  
while ( !done ) {  
    EventType type;  
    int x;  
    int y;  
    char keystroke;  
    mainWindow.getNextEvent( type, x, y, keystroke );  
    switch ( type ) {  
    case LeftPress:     cout << "left press"; break;  
    case LeftRelease:   cout << "left release"; break;  
    case MiddlePress:   cout << "middle press"; break;  
    case MiddleRelease: cout << "middle release"; break;  
    case RightPress:    cout << "right press"; break;  
    case RightRelease:  cout << "right release"; break;  
    case MouseMotion:   cout << "mouse motion"; break;  
    case KeyStroke:     cout << "keystroke"; break;  
    case WindowRefresh: cout << "window refresh"; break;  
    default:            cout << "unknown event"; break;  
    }  
    cout << at( "at( " << x << "," << y << ")).char = ";  
    if (32 < keystroke && keystroke < 127) cout << keystroke;  
    else if (keystroke == 32) cout << "<blank>";  
    else cout << "(ascii)" << int( keystroke );  
    cout << endl;  
    if ( KeyStroke == type && 'Q' == keystroke ) done = true;  
}  
}  

Figure C.7: An event loop for a program that uses sxwin.
Appendix D

Useful Unix Utilities

Each of the following utilities is useful for maintaining software. The best way to obtain additional information is man utility, where utility is the name of the utility. A good reference on Unix commands and utilities is Unix in a Nutshell [Gil92].

a2ps Creates “pretty” printout of your programs to turn in or debug; on Solaris, you need to do add psutils or

    alias a2ps /afs/eos.ncsu.edu/contrib/psutils/bin/a2ps

in order to get the most current version (put either command into your .mycshrc file). Since a2ps does not have a man page, Figure D.1 shows the most useful options. Usage is a2ps options files, where options is a list of options and files is a list of files. Linux users can accomplish the same things (and more) with the enscript utility.

diff Reports differences between two given files. If you keep old versions of source files (a good idea), diff can tell you what changes you made to fix a bug (or introduce a new one).

foreach-end (C-shell only) Repeats a sequence of commands for every file (item) in a given list. For example, the following adds a comment with the file name, my name and the date to the beginning of each source file:

    unity% foreach file ( *.cpp *.h )
    foreach? touch $file.tmp
    foreach? echo "/// $file -" >> $file.tmp
    foreach? echo "/// Matt Stallmann,‘date‘ >> $file.tmp
    foreach? mv $file.tmp $file
    foreach? end

grep (get regular expression and print) Searches for occurrences of particular strings and patterns in a list of files. Can be used to locate all references to a particular variable or function. If you’re only searching for a string fgrep is much faster than grep.

head Gives the first $n$ lines of a file, where $n = 10$ (default) or specified by a command-line switch. (see also tail)
<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>-1</code> (&quot;one&quot;)</td>
<td>print one logical page per physical page instead of the usual 2 (numbers from 1-9 can be used, but anything beyond 2 is probably too hard to read).</td>
</tr>
<tr>
<td><code>-A</code></td>
<td>print more than one file per page - usually each new file begins on a new (physical) page.</td>
</tr>
<tr>
<td><code>-E language</code></td>
<td>pretty print assuming the input is in the given language (standard extensions, such as <code>.cpp</code> are recognized automatically, but you may want to use <code>-E cpp</code> to print <code>.h</code> files — <code>a2ps</code> assumes C rather than C++).</td>
</tr>
<tr>
<td><code>-l n</code> (&quot;ell&quot;)</td>
<td>print <code>n</code> characters per line (the font will be scaled accordingly).</td>
</tr>
<tr>
<td><code>-L n</code></td>
<td>print <code>n</code> lines per page (again, the font will be scaled).</td>
</tr>
<tr>
<td><code>-o name</code></td>
<td>send PostScript output to the file <code>name</code>.</td>
</tr>
<tr>
<td><code>-T n</code></td>
<td>set tab stops at every <code>n</code> spaces; especially useful if your editor has tabs set at less than 8 spaces.</td>
</tr>
</tbody>
</table>

Figure D.1: Options for the `a2ps` utility.
**nm** (name list) Used to print the symbol table for an object file:

```
csc% nm averages.o
000002b8 t Letext
00000280 r MAX_SCORES
   U _Q_qtod
00000000 ? __FRAME_BEGIN__
   U __ls__7ostreamPCc
   U __ls__7ostreamPFR7ostream_R7ostream
   U __ls__7ostreamf
   U __rs__7istreamRi
   U __throw
   U cin
   U cout
   U endl__FR7ostream
00000000 T main
   U sort__FPii
```

**od** (octal dump) Shows the exact, byte-for-byte, contents of a file. `od -c` is a useful way to see if a file has any hidden garbage in it.

**tail** Gives the last $n$ lines of a file, where $n = 10$ (default) or specified by a command-line switch. Can also be used to give the middle part of a file (see the man page).

**touch** Creates an empty file with the given name, or changes the modification date of the file if it exists.

### Special features of the C-shell

(similar features exist in other shells)

**the -r switch** (all shells) Some commands, such as `cp`, `diff`, and `rm`, can be executed *recursively* (applied to all subdirectories) using the `-r` switch. For example, `rm -r dead_dir` removes the directory `dead_dir` and all of its files and subdirectories (and all their files and subdirectories in turn).

**the quote marks: ’, ”, and ‘** (C-shell only) Any string within any pair of quotes is treated as a single word (argument) by the shell.

Single quotes (‘) are used to surround strings that need to be protected from shell interpretation. For example, you can say

```
unity% grep 'x * y' *.cpp *.h
```

to look for all occurrences of the string `$x * y$` in source files.

Double quotes (""`) are the same as single quotes, except that variable substitution can occur inside them. For example,

```
```
unity% set today = '28 July 1999'
unity% grep "MODIFIED $today" *.cpp *.h

finds all occurrences of MODIFIED 28 July 1999 in the source files.
Back quotes ('') enclose strings that are to be treated as separate commands. The output of
the given command is then used as a string argument to the larger command. For example,

unity% grep 'head -1 main.cpp' *.cpp *.h

finds all occurrences of the first line of main.cpp in the source files. (do man csh for more
information).

**input/output redirection** If a program is written to read data from standard input (cin) and
write data on standard output (cout), either or both can be redirected from or to a file. Simple
redirection works as follows:

```
prog > file   send output of program prog to file file
prog >> file  append output of program prog to file file
prog < file   take input of program prog from file file
```

Of course you can redirect both input and output simultaneously as in

```
prog < file1 > file2
```

You can also redirect standard error (cerr) output as follows:

```
prog >& file  send error output of program prog to file file
prog >>= file append error output of program prog to file file
(prog > file1) >& file2 send standard output to file1, error output to file2
```

Finally, you can use a pipe (|) to send the output of one program to the input of another (and
|& works analogously with standard error). For example,

```
% wc *.cpp *.h | sort -n
```

produces a list of source files in the current directory, sorted by increasing number of lines.

**using - as an argument** Some Unix commands, such as rm use the argument - to specify that
the next argument is to be interpreted as a file name no matter what. For example,

```
% rm - f
```

allows you to remove a file whose name (by some unfortunate accident) happens to be -f.
Other commands use the argument - to denote standard input. For example,

```
% cat file1 - file2 > file3
```

puts what you type on the xterm (standard input) between file1 and file2 and stores the
result in file3.
Appendix E

Reference Guide

Address-of-operator (&). The C++ operator which returns the address of an object.

Alias. An alternate name for an object.

argv/argc. see Command Line Arguments

ASCII Character set. See Figure E.1

assert “macro” (assert.h). Used to test a condition, e.g., assert( x > 0 ); If the condition is not true an error message is output and the program is aborted.

```cpp
#include <assert.h>
#include <iostream.h>

void sub(int x)
//PRE: x > 0
{
    assert(x > 0);
}

int main()
{
    sub(-5);
}

csc% a.out
testassert.cpp:7: failed assertion ‘x > 0’
Abort
```

Automatic Memory Allocation. Memory allocated at the start of a function call and deallocated when the function returns. Used to hold local variables.

Bus Error. Generally occurs when a pointer is not initialized and then used to access data in memory with an address that is not valid for the data’s type. What it really means is that an
<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
</tr>
</thead>
<tbody>
<tr>
<td>nul</td>
<td>soh</td>
<td>stx</td>
<td>etx</td>
<td>eot</td>
<td>enq</td>
<td>ack</td>
<td>bel</td>
<td>bs</td>
<td>ht</td>
</tr>
<tr>
<td>nl</td>
<td>vt</td>
<td>ff</td>
<td>cr</td>
<td>so</td>
<td>si</td>
<td>dle</td>
<td>dcl</td>
<td>dc2</td>
<td>dc3</td>
</tr>
<tr>
<td>dc4</td>
<td>nak</td>
<td>syn</td>
<td>etb</td>
<td>can</td>
<td>em</td>
<td>sub</td>
<td>esc</td>
<td>fs</td>
<td>gs</td>
</tr>
<tr>
<td>rs</td>
<td>us</td>
<td>sp</td>
<td>!</td>
<td>&quot;</td>
<td>#</td>
<td>$</td>
<td>%</td>
<td>&amp;</td>
<td>'</td>
</tr>
<tr>
<td>(</td>
<td>)</td>
<td>*</td>
<td>+</td>
<td>,</td>
<td>-</td>
<td>.</td>
<td>/</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td>9</td>
<td>:</td>
<td>;</td>
</tr>
<tr>
<td>&lt;</td>
<td>=</td>
<td>&gt;</td>
<td>?</td>
<td>@</td>
<td>A</td>
<td>B</td>
<td>C</td>
<td>D</td>
<td>E</td>
</tr>
<tr>
<td>F</td>
<td>G</td>
<td>H</td>
<td>I</td>
<td>J</td>
<td>K</td>
<td>L</td>
<td>M</td>
<td>N</td>
<td>O</td>
</tr>
<tr>
<td>P</td>
<td>Q</td>
<td>R</td>
<td>S</td>
<td>T</td>
<td>U</td>
<td>V</td>
<td>W</td>
<td>X</td>
<td>Y</td>
</tr>
<tr>
<td>Z</td>
<td>[</td>
<td>\</td>
<td>]</td>
<td>^</td>
<td>_</td>
<td>`</td>
<td>a</td>
<td>b</td>
<td>c</td>
</tr>
<tr>
<td>d</td>
<td>e</td>
<td>f</td>
<td>g</td>
<td>h</td>
<td>i</td>
<td>j</td>
<td>k</td>
<td>l</td>
<td>m</td>
</tr>
<tr>
<td>n</td>
<td>o</td>
<td>p</td>
<td>q</td>
<td>r</td>
<td>s</td>
<td>t</td>
<td>u</td>
<td>v</td>
<td>w</td>
</tr>
<tr>
<td>x</td>
<td>y</td>
<td>z</td>
<td>{</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>~ del</td>
</tr>
</tbody>
</table>

Figure E.1: ASCII Character set.
address is “out of alignment” — for example, if the address used to refer to an int is not divisible by 4 (since an int takes up 4 bytes it must begin at the beginning of a 4-byte chunk of memory).

C++ Identifier. A series of characters consisting of letters, digits, and underscores (_) that does not begin with a digit.

C++ Keywords.

auto, asm, bool, break, case, catch, char, class, const, const_cast, continue, default, delete, do, double, dynamic_cast, else, enum, explicit, extern, false, float, for, friend, goto, if, inline, int, long, mutable, namespace, new, operator, private, protected, public, register, reinterpret_cast, return, short, signed, sizeof, static, static_cast, struct, switch, template, this, throw, true, try, typedef, typeid, typename, union, unsigned, using, virtual, void, volatile, wchar_t, while

C-style string. An array of characters ending in the null character (‘\0’)

char string1[] = "Hello";
const int SIZE = 4;
char string2[SIZE] = {'a', 'b', 'c', '\0'};
char* string3 = "Goodbye";
char* string4 = new char[SIZE];
strcpy(string4, "def");

Character Handling Library (ctype.h). Provides testing and manipulation of characters

(For more info: eos% man ctype)
#include <ctype.h>
char ch;

//character testing
if (isdigit(ch)) ....    //Is ch a digit (0-9)?
if (isalpha(ch)) ....    //Is ch a letter (A-Z, a-z)?
if (isalnum(ch)) ....    //Is ch a letter or digit(0-9, A-Z, a-z)?
if (islower(ch)) ....    //Is ch a lowercase letter(a-z)?
if (isupper(ch)) ....    //Is ch an uppercase letter(A-Z)?
if (isspace(ch)) ....    //Is ch a white-space character?:
    // newline(\n)
    // space( ' ' )
    // form feed(\f)
    // carriage return(\r)
    // horizontal tab(\t)
    // vertical tab(\v)

//character manipulation
ch = (tolower('A'))      //Convert ch to lowercase letter
ch = (tolower('a'))      //Don't change ch if it's already a lowercase letter
ch = (toupper('a'))      //Convert ch to uppercase letter
ch = (toupper('A'))      //Don't change ch if it's already an uppercase letter

Command Line Arguments (argv, argc).

#include <iostream.h>

int main(int argc, char* argv[]) //argc - number of arguments on command line
    // including program name
    //argv[] - array of character strings which
    // contain arguments
    // DO NOT MODIFY THE argv[] ARRAYS!
{
    cout << "argc = " << argc << endl;

    for (int i = 0; i < argc; i++)
        cout << "argv[" << i << "] = " << argv[i] << endl;
}

csc% a.out file1 file2 file3
argc = 4
argv[0] = a.out
argv[1] = file1
argv[2] = file2
argv[3] = file3

Do not attempt to modify the argv[] arrays!
**const Call-by-value.** A const parameter in a function list cannot be changed in the body of the function:

```cpp
c void sub(const int i)  
{  
i += 5;    //illegal!!
}
```

c++ testconst.cpp

testconst.cpp: In function ‘void sub(int)’:  
testconst.cpp:3: assignment of read-only parameter ‘int const i’

**const Class Member Function.** A const member function cannot change any of the class data. It is the only type of function that can be called on a const class object. (see **const Class Object** for example)

**const Class Object.** A const class object must be initialized when it is declared and its value cannot be changed. Only class member functions declared as const may be used with const class objects:

```cpp
int getMonth() const;
...
const Date newYears(1,1);
cout << newYears.getMonth();
```

**const Reference.** A const reference to a variable or object may be passed to a function. The function will be able to directly access the variable/object, but will not be permitted to modify it. A const reference to a variable/object may be returned from a function which gives the caller direct access to the variable/object, but not the ability to modify it:

```cpp
int squareNumber(const int & number);

void printTheDate(const Date & d);

const Student & searchList(const ClassRoll & roll, const char * name);
```

**const Variable.** A const variable must be initialized when it is declared and its value cannot be changed:

```cpp
const int NUMBER = 5;

const float INTEREST = 5.0;

const int MAX_SIZE = 20;
```

**Corrupted Heap.** A heap which has been damaged by an improper delete, or by writing to an object after it has been deleted. Typically causes a segmentation fault.

**Dangling Pointer.** A (non-NULL) pointer which doesn’t point to a valid object.
delete Operator. The C++ dynamic memory deallocation operator.

Deference Operator (*). The C++ operator which operates on a pointer and fetches the referent (object pointed to by the pointer).

Destructor. A special member function which is called when an object goes out of existence and is responsible for deleting all dynamically allocated memory used by the object.

do-while loop (control structure).

do {
...
} while (condition);

Escape Sequences.

\' - Single quote
" " - Double quote
\ - Backslash
\a - Audible alert (bell)
\b - Backspace
\f - Formfeed
\n - Newline
\t - Horizontal tab
\v - Vertical tab

Dynamic Memory Allocation. Runtime memory allocation using new and delete.

Dynamic Object. An object whose memory is dynamically allocated.

File I/O (fstream.h).

#include <fstream.h>

int main()
{
char ch;
ifstream in("file.in"); //Open file.in for input
ofstream out("file.out"); //Open file.out for output
if (in) {
  //If file.in open successful
  while (in.get(ch)) // while not at eof, read in
    out << ch; // a char and output to file.out
  in.close(); // Close file.in
}
else
  cout << "Error opening file.in" << endl;
out.close(); //Close file.out
}

for loop (control structure).
for (expression1; expression2; expression3) {
    ...
}

- **expression1** - optional initialization statement(s), generally used to initialize loop control variable
- **expression2** - loop continuation condition
- **expression3** - optional statement(s) that are executed at the end of each loop generally used to increment/decrement loop control variable

The sequence `for ( A; B; C ) D`, where D is either a statement or a block, is identical to `{A; while ( B ) { {D} C; }}`.

```cpp
const int MAX = 5;
for (int i = 0; i < MAX; i++) { // variable i exists only within for loop
    ...
}
int j;
for (j = 10; j > 0; j--) { // variable j exists outside for loop
    ...
}
bool done = false;
for (int i = 1, j = 0; i <= MAX & & !done; i += 2, j++){
    ...
}
```

Node* tmp = myHead;
for (; tmp->myLink ; tmp = tmp->myLink); // Move tmp to last node in linked list

**friend Keyword.** Used within a class declaration to declare a function or another class as a friend of the class. The friend class/function has access to the private members of the class. SHOULD NOT BE USED UNLESS ABSOLUTELY NECESSARY!

```cpp
class Student {
    friend ostream& operator<<(ostream&, const Student&);
    friend class StudentList;
    ...
};
```

**Indirection Operator ().** Same as the dereference operator.

**inline Keyword.** Suggests to the compiler that the code for the function be inserted anywhere that the function is called
inline int square(int n)
{
    return n*n;
}

A class function that is defined in the class declaration is automatically inlined.

class Number {
    public: Number(int n);
        int getNumber() { return myNumber; } //inline class function
    private: int myNumber;
};

Input/Output Stream Manipulation (iomanip.h). Some stream manipulators, like setw (sets the field width for the next value) and setprecision (sets the precision for floating point numbers) are relatively straight-forward. Others involve setting and resetting the ios flags as shown in Figures E.3 and E.2.

Lexicographic. We say that a list of strings is sorted lexicographically if they are essentially in dictionary order (the rules may vary according to how upper-case versus lower-case, non-alphabetic characters, and other issues are treated). In most CSC 214 situations, the correct technical definition is that strings are sorted lexicographically if, whenever string1 precedes string2, the value of strcmp(string1,string2) is ≤ 0.

Lost Memory. Memory lost due to a memory leak. You could also call it leaked memory.

Memory Leak. Occurs when a program fails to deallocate dynamic memory that is no longer accessible from any existing pointers.

Null Pointer. A pointer whose value is NULL (typically address 0). By convention, nothing is stored at address NULL, so that NULL is used as a way of “pointing at nothing.”

Pointer. The address of an object.

Pointer Data Object. An object whose value is a pointer.

| Number(123.456) in 10 char wide field using default settings: 123.456 |
| Number(123.456) in scientific notation: 1.234560e+02 |
| Number(123.456) in fixed format with 2 decimal places: 123.46 |
| Number(123.456) left-justified in 10 char wide field: 12.346 |
| Number(123.456) right-justified in 10 char wide field: 123.46 |
| Number(123.456) with leading + sign: +123.46 |
| Another number(10) with decimal point but no decimal places: +10. |
| Characters right-justified with * fill char: ********abcdef123456 |
| Characters left-justified with # fill char: abcdef123456####### |
| Characters right-justified with - fill char: ---------abcdef123456 |

Figure E.2: Output from stream manipulation program.
#include <iomanip.h>
#include <iostream.h>

int main()
{
    float number = 123.456;
    cout << "Number(123.456) in 10 char wide field using default settings: 
    << setw(10) << number << endl;
    cout << setiosflags(ios::scientific) << "Number(123.456) in scientific notation: 
    << number << endl;
    cout << resetiosflags(ios::scientific) << setiosflags(ios::fixed) 
        << setprecision(2) << "Number(123.456) in fixed format with 2 decimal places: 
    << number << endl;
    cout << setiosflags(ios::left) 
        << "Number(123.456) left−justified in 10 char wide field: 
    << setw(10) << number << endl;
    cout << resetiosflags(ios::left) 
        << "Number(123.456) right−justified in 10 char wide field: 
    << setw(10) << number << endl;
    cout << setiosflags(ios::showpos) << setiosflags(ios::fixed) 
        << "Number(123.456) with leading + sign: 
    << number << endl;
    float number1 = 10;
    cout << setiosflags(ios::showpoint) << setprecision(0) 
        << "Another number(10) with decimal point but no decimal places: 
    << number1 << endl;
    char* characters = "abcdef123456*";
    cout << "Characters right−justified with * fill char: 
    << setfill('*') << setw(20) << characters << endl;
    cout << setiosflags(ios::left) 
        << "Characters left−justified with # fill char: 
    << setfill('#') << setw(20) << characters << endl;
    cout << resetiosflags(ios::left) 
        << "Characters right−justified with − fill char: 
    << setfill('-') << setw(20) << characters << endl;
}
Redirect of Input/Output. Ability provided by UNIX for standard input(cin) to come from a file rather than the keyboard and standard output(cout) to go to a file rather than the screen. With the simple program:

```cpp
#include <iostream.h>
int main() {
    char ch;
    while (cin.get(ch))
        cout << ch; }
```

you can get the following variations:

- `csc% a.out` Input from keyboard, output to screen
- `csc% a.out < file.in` Input from file.in, output to screen
- `csc% a.out > file.out` Input from keyboard, output to file.out
- `csc% a.out < file.in > file.out` Input from file.in, output to file.out

Reference. A type which represents the name (not address) of an object. A reference to an object is an alias for the object.

Referent. The object pointed to by a pointer.

Static Memory Allocation. Memory which is allocated when the program starts (before main is called) and isn’t deallocated until the program halts.

Segmentation Fault. Occurs when a program tries to access a memory location with an address outside the “segment” it’s been given; usually the result of going far beyond the bounds of an array or trying to dereference a null pointer.

String Handling Library (string.h). Provides functions for searching, manipulating, comparing, and determining the length of C-style strings (For more info: `eos% man string`)

```cpp
#include <string.h>
const int MAX = 30;
char s1[MAX] = "abc";
char s2[MAX] = "def";
int size = strlen(s1);  //Determine length(number of characters preceding "\0" character) of string s1
strcat(s1,s2);   //Concatenate string s2 to string s1
strcpy(s1,s2);   //Copy string s2 over string s1
int comp = strcmp(s1,s2); //Compare string s1 to string s2
    // comp < 0 if s1 < s2
    // comp = 0 if s1 = s2
    // comp > 0 if s1 > s2

switch (decision statement, control structure).
```
#include <iostream.h>
int main()
{
    enum {BEGINNING, MIDDLE, END};
    const int MAX = 3;
    int i = 0;
    while (i <= MAX) {
        switch (i) {
        case BEGINNING: cout << "Beginning" << endl;
                        break;
        case MIDDLE:    cout << "Middle"  << endl;
                        break;
        case END:       cout << "End"     << endl;
                        break;
        default:        cout << "SHOULD NOT BE HERE" << endl;
                        break;
        }
        i++;
    }
}

csc% a.out
Beginning
Middle
End
SHOULD NOT BE HERE

Ternary Conditional Operator (?:) (decision statement, control structure).

    condition ? true expression : false expression;

    x < y ? y = 100 : y = 0;  // If x is less than y, set y to 100
    // Else set y to 0
    cout << (x < y ? "x is less than y" : "x is not less than y") << endl;

while loop (control structure).

    while (condition) {
        ...
    }
Index

-o switch, caution, 41
.cpp.o, 45
$*, 45

a.out, 35
a2ps, 185
abstract base class, 153, 161
activation record, 117, 138
address, 75, 87
aggregate type, 176
algorithm, 11, 31
alias, 31
all (makefile target), 44
allocation
    automatic, 189
    dynamic, 194
    static, 198
ambiguates, 175
ANSI, 175
ascii, 189
assert, 109, 189
assignment
    used as truth value, 176
atof, 56
automatic memory allocation, 90

base operand, 175
binary number, 56
black box testing, 24
black-box testing, 34
break point, 177
bubble diagram, 49, 60
build, 41
bus error, 189

C preprocessor, 40
c++filt, 39, 173

c-String, 191
cascading, 65
class, 61, 73
    calling member functions, 63, 74
    const functions, 63, 74
    constructor, 62, 73
    default constructor, 62, 73
    function definition syntax, 64, 73
    initialization list, 70, 74
    null constructor, 62, 73
    preprocessor directives (ifdef, define, endif), 64, 74
    private class members, 62, 73
    public class members, 61, 73
classes
    with dynamically allocated data, 82, 90
client, 21, 61
client programmer, 40
code reuse, 47
command line arguments, 192
compilation, 47
compilation and linking (diagram), 42
compile time, 35
compiler options, 47
crude classes, 153
const, 70, 193
constructor, 62, 73
contract programming, 21, 31
copy constructor, 85
cryptic messages, 39
dangling pointer, 78, 90
debugger, 176
debugging printout, 24, 34
deep copy, 85, 91
default constructor, 62, 73
delete operator, 78, 90
INDEX

dependency, 41
  rules, 43, 45
destructor, 83
diff, 185
Don't know how..., 44
dynamic binding, 152
dynamic memory allocation, 90
enscript, 185
error, 35
  compile time, 35
  link time, 38
  symbol referencing, 38
effects
  makefile, 44
escape sequences, 194
evaluating polynomials, 56
executable program, 41
fgrep, 185
file I/O, 194
finite-state machine, 49, 57
  bubble diagram, 49, 60
  examples, 50-52, 54, 58
  Horner's Rule, 56, 60
  state, 49, 57
  table, 53, 54
  text, 51, 52, 54
  text-processing, 51, 60
  transition, 49, 60
  transition table, 52, 60
  while-switch idiom, 53, 60
friend, 195
function call
  matching, 175
gdb, 176
grep, 51, 185
header files, 40
Horner's Rule, 56, 58, 60
implicit declaration, 38
industrial-strength testing, 23
inheritance, 157
inheritance hierarchy, 140
initialization list, 70, 74
inline, 70, 195
input/output redirection, 198
instance, 14
iomanip, 196
isalpha, 191
isdigit, 191
keywords, 191
lacks a cast, 37
legacy C programs, 37
lexicographic, 196
linked list, 137
  array (contrast), 105
  traversal, 137
linker, 38
linking, 35, 47
loader, 38
loading, 35
lost memory, 196
macro, 45
magic number, 175
make, 45
  all, 44
  clean, 44
  gnu, 46, 47
make and makefile, 41
makefile
  example of, 43, 46, 47
mangled names, 173
memory leak, 82, 90, 196
methods, 14
modification time, 43
multiple inheritance, 140
name lookup, 175
name mangling, 38, 39, 70, 173
new operator, 77, 90
nm, 38, 174, 187
no match, 175
Node, 137
non-pointer type, 175
NULL, 196
null constructor, 62, 73
null pointer, 196
object file, 35
Object-oriented design, 13
object-oriented design, 34
overloaded, 70
overloaded assignment operator, 86

parse error, 175
pipes, 188
pointer, 75, 87
Polymorphism, 146
polymorphism, 161
post-conditions, 22
postconditions, 31
preconditions, 22, 31
preprocessing, 47
pretty printing, 185
private class members, 62, 73
protected, 142
prototype
  does not match, 176
prototypes, 20
pseudo-code, 11
public class members, 61, 73
public inheritance, 142
Pure virtual function, 161
pure virtual function, 153

real number, 49
recursion, 112, 137
redefinition, 176
 redirection, 188
 redirection, I/O, 198
removing
  file with funny name, 188
run-time stack, 117
runtime stack, 138

scope, 65, 70
scope-resolution operator, 145
scoping, 175
search, 51
segmentation fault, 198
setprecision, 196
setw, 196
shallow copy, 85, 90
signature, 70, 153

single inheritance, 140
slicing, 146, 157
soda machine, 50
source file, 35
stack, 117
state, 49, 57
static
  function, 102
static binding, 150
static memory allocation, 87
Stepwise refinement, 12
stepwise refinement, 12
string library, 198
struct, 71
structure data type, 71
stubs, 23
symbol reference, 38
symbol table
  of a .o file, 174
syntax error, 176
tail recursion, 118
target, 41
  dummy, 44
generic, 45
  up-to-date, 43
test coverage, 23
test driver, 24
testing
  black box, 24
  white box, 23
text-processing, 51, 60
this pointer, 86
top-down design, 12
top-down design with stepwise refinement, 31
touch, 187
toupper, 191
transition, 49, 60
transition table, 53, 60
type specification
  invalid, 176
undefined function, 176
undefined symbol, 38
Unexpected end of line, 45
virtual function, 152, 161
INDEX

waterfall model, 17, 34
while-switch idiom, 53, 60
white box testing, 24
white-box testing, 34
word count, 52
Bibliography


