## Contents

Preface vii

1 Abstraction 1
   1.1 A Pay Calculator ........................................ 1
   1.2 Design ................................................... 4
   1.3 Names .................................................... 8
   1.4 Implementation ......................................... 9
   1.5 Modularity and Abstraction .............................. 16

2 Data Abstraction 21
   2.1 Introduction ............................................ 21
   2.2 Classes to Enforce Data Abstraction ................. 23
   2.3 Classes to Support Object-Oriented Programming .... 24
   2.4 Constant Methods ........................................ 32
   2.5 Inline Methods .......................................... 33
   2.6 Constructors ............................................ 36
   2.7 Get and Set Methods ..................................... 40
   2.8 Operators ................................................ 44
   2.9 Compiling Large Programs ............................... 50
   2.10 The make Utility ........................................ 54

3 Strings and Streams 57
   3.1 C Strings ................................................. 57
   3.2 C++ Strings ............................................... 61
   3.3 I/O Streams .............................................. 65
   3.4 String Streams ........................................... 69
# CONTENTS

4 Error Checking .................................................. 73
   4.1 Introduction ................................................. 73
   4.2 Exceptions .................................................. 76
   4.3 Input Validation ............................................. 79

5 Vectors .......................................................... 87
   5.1 A Simple File Viewer ........................................ 87
   5.2 Arrays and Vectors .......................................... 88
   5.3 Vectors in the STL ........................................... 91
   5.4 Implementation of the File Viewer ......................... 93

6 Dynamically Allocated Arrays ................................. 99
   6.1 The Size of Ordinary Arrays ............................... 99
   6.2 The Dynamic Allocation of Arrays ......................... 100
   6.3 Programming with Dynamically Allocated Arrays ......... 102

7 Implementation of Vectors .................................... 109
   7.1 Generic Programming ....................................... 109
   7.2 A Basic Class of Vectors .................................. 112
   7.3 Some Additional Methods .................................. 117
   7.4 Destroying and Copying Vectors ........................... 119

8 Analysis of Algorithms ..................................... 127
   8.1 Introduction ................................................ 127
   8.2 Measuring Exact Running Times ............................ 128
   8.3 Analysis ..................................................... 130
   8.4 Asymptotic Running Times ................................. 132
   8.5 Some Common Running Times ............................... 135
   8.6 Basic Strategies ............................................ 137
   8.7 Worst-Case and Average-Case Analysis .................... 143
   8.8 The Binary Search Algorithm ............................... 146

9 Linked Lists .................................................... 151
   9.1 A Simple Text Editor ....................................... 151
   9.2 Linked Lists in the STL .................................... 155
   9.3 Design and Implementation of the Text Editor ............ 161
## CONTENTS

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>10 Maps</strong></td>
<td></td>
</tr>
<tr>
<td>10.1 A Phone Book</td>
<td>167</td>
</tr>
<tr>
<td>10.2 Maps in the STL</td>
<td>170</td>
</tr>
<tr>
<td>10.3 Design and Implementation of the Phone Book</td>
<td>176</td>
</tr>
<tr>
<td><strong>11 Object-Oriented Design</strong></td>
<td>187</td>
</tr>
<tr>
<td>11.1 The Software Life Cycle</td>
<td>187</td>
</tr>
<tr>
<td>11.2 The Software Development Process</td>
<td>190</td>
</tr>
<tr>
<td>11.3 Specification</td>
<td>192</td>
</tr>
<tr>
<td>11.4 Design</td>
<td>196</td>
</tr>
<tr>
<td>11.5 Implementation</td>
<td>208</td>
</tr>
<tr>
<td><strong>12 Generic Algorithms</strong></td>
<td>213</td>
</tr>
<tr>
<td>12.1 A Generic Counting Algorithm</td>
<td>213</td>
</tr>
<tr>
<td>12.2 Iterators</td>
<td>218</td>
</tr>
<tr>
<td>12.3 Generic Algorithms in the STL</td>
<td>221</td>
</tr>
<tr>
<td><strong>13 Implementation of Linked Lists</strong></td>
<td>227</td>
</tr>
<tr>
<td>13.1 Nodes and Links</td>
<td>227</td>
</tr>
<tr>
<td>13.2 Some Basic Methods</td>
<td>229</td>
</tr>
<tr>
<td>13.3 Iterators, Insert and Erase</td>
<td>233</td>
</tr>
<tr>
<td>13.4 Destroying and Copying Linked Lists</td>
<td>240</td>
</tr>
<tr>
<td><strong>14 Recursion</strong></td>
<td>245</td>
</tr>
<tr>
<td>14.1 The Technique</td>
<td>245</td>
</tr>
<tr>
<td>14.2 When to Use Recursion</td>
<td>252</td>
</tr>
<tr>
<td>14.3 Tail Recursion</td>
<td>255</td>
</tr>
<tr>
<td><strong>15 Sorting</strong></td>
<td>257</td>
</tr>
<tr>
<td>15.1 Selection Sort</td>
<td>257</td>
</tr>
<tr>
<td>15.2 Insertion Sort</td>
<td>261</td>
</tr>
<tr>
<td>15.3 Mergesort</td>
<td>264</td>
</tr>
<tr>
<td>15.4 Quicksort</td>
<td>268</td>
</tr>
<tr>
<td><strong>A Library Reference</strong></td>
<td>275</td>
</tr>
<tr>
<td>A.1 C Strings</td>
<td>275</td>
</tr>
<tr>
<td>A.2 C++ Strings</td>
<td>277</td>
</tr>
<tr>
<td>A.3 I/O Streams</td>
<td>284</td>
</tr>
<tr>
<td>A.4 String Streams</td>
<td>288</td>
</tr>
</tbody>
</table>
CONTENTS

A.5 Vectors ...................................................... 288

Bibliography ..................................................... 291
Preface

These notes are for a second course on computer programming and software development. In a first course, you likely focused on learning the basics: variables, control statements, input and output, files, arrays and functions. These concepts are critically important and they are sufficient for the creation of many useful programs. But many other programs, especially large ones, require more powerful concepts and techniques.

The creation of large computer programs poses three basic challenges. First, a large program always contains a large amount of details and all this information can be difficult to manage. Second, a large program typically holds a large amount of data that must organized so it can be accessed efficiently. Third, a large program often performs complex tasks that require nontrivial techniques. The overall goal of these notes is to teach you concepts and techniques that will allow you to meet these three challenges. Here’s an overview.

First, the construction of a large program becomes more manageable if the program is organized into a set of smaller components. This modularity is usually achieved through the technique of abstraction. Even if you are not familiar with these terms, you have already used abstraction in your programs: abstraction happens automatically every time you create a function. In these notes, you will learn how to apply the technique of abstraction to data.

Second, to store large amounts of data, computer scientists have invented a wide variety of data structures that allow data to be stored and accessed efficiently. You are already familiar with one data structure: the array. In these notes, you will learn about these other basic data structures: vectors, linked lists, sets and maps. You will learn how and when to use these data structures. You will also learn the techniques involved in the implementation of the first two.

Third, computer scientists have also designed algorithms that perform com-
plex tasks efficiently. In these notes, you will learn efficient algorithms for searching and sorting. You will also learn to design algorithms using the technique of recursion and how to analyze the efficiency of simple algorithms.

Most of the concepts and techniques you will learn will be introduced through the creation of programs that are representative of real-life software. These examples will show that these concepts and techniques are tools that have been developed to solve real problems.

As you learn about data abstraction, data structures and algorithms, you will also learn about a number of other important topics such as the overall process of developing software, the importance of good documentation, object-oriented design, classes, pointers, dynamic memory allocation, exceptions, testing, the use of standard software components (as available in a standard library), and the creation of generic software components using templates and iterators.

These notes use the programming language C++ and its associated Standard Template Library (STL). Even though the main focus of these notes is not on the C++ language itself, you will learn all the relevant features of C++ and the STL. In particular, you will learn to implement data structures that are realistic subsets of the data structures provided in the STL.

After reading these notes and working on the exercises, you will be able to create fairly complex computer programs. But you will also be prepared to continue your study of computer science and software development. For example, at Clarkson University, these notes were written for CS142 Introduction to Computer Science II, a course that leads directly to CS242 Advanced Programming Concepts (graphical user interfaces, more advanced object-oriented programming), CS344 Algorithms and Data Structures (more advanced than those covered in these notes) and CS241 Computer Organization. In fact, the topics covered in these notes form the foundation you for the study of almost every other subject in computer science, including programming languages, operating systems, artificial intelligence, cryptography, computer networks, database systems, as well as, of course, large-scale software development.

Feedback on these notes is welcome. Please send comments to alexis@clarkson.edu.
Chapter 1
Abstraction

In this chapter, we will create a relatively simple program. This will allow us to review basic programming concepts such as files, functions and structures. It will also allow us to discuss the important topics of modularity and abstraction.

1.1 A Pay Calculator

We will create a pay calculator that computes what each employee of a small company should be paid for a day of work. The program reads a file containing the times when the employees started and stopped working. The program then computes the amount each employee should be paid and writes that information to another file.

The creation of a program such as this pay calculator involves several activities. The first one is the specification of the software. This consists in determining exactly what the software must do, from the point of view of the user. In other words, a specification describes the external behavior of the software (not its internal workings). A good specification should be clear, correct and complete. It also helps if it is as concise as possible. The specification of a program normally involves communicating with the client (or with potential users).

Figures 1.1 and 1.2 show the specification of a first version of our pay calculator. Note, in particular, how the input and output formats are described in detail.

Note also that this specification is only for a first version of the program. It is difficult to create a very large program in one shot. It is usually much
Pay calculator
Version 1

SPECIFICATION

OVERVIEW

This program computes what each employee of a company should be paid for a day of work. The program reads a file containing start and stop times for each employee. The program prints the pay amounts to another file.

DETAILS

The input file is called "times.txt" and located in the same folder as the program. The file contains one line for each employee. Each line contains an employee number, a start time and a stop time. These three items are separated by blank spaces.

Employee numbers are positive integers. Times are given on a 24-hour clock (0:00 to 23:59) in the format h:mm or hh:mm, where each h and m stands for a single digit.

For example, "17 8:00 16:30" means that employee 17 started working at 8:00 a.m. and stopped working at 4:30 p.m.

The lines in the input file are sorted in increasing order of employee number.
The output file is called "report.txt" and located in the same folder as the input file and program. The output file contains one line for each employee. Each line consists of an employee number followed the start time, stop time and pay amount for that employee. These four items are separated by a single space. The times are printed in the same format as in the input file. The pay amount is printed with a dollar sign and exactly two digits after the decimal point.

For example, "17 8:00 16:30 $104.00" means that employee 17 worked from 8:00 a.m. to 4:30 p.m. and should be paid $104.

The lines in the output file are sorted in increasing order of employee number (as in the input file).

All employees are paid $12 per hour. The pay amounts are computed as exactly as possible.

No error-checking is performed in this version of the program.
CHAPTER 1. ABSTRACTION

It is easier to develop it gradually by building successfully more complete versions. This is called **incremental** (or **iterative**) development.

Incremental development has several significant advantages. First, the experience and knowledge gained in building one version of the software can be used in the building of the following versions. Second, it is possible to get feedback from the client on early versions. This helps to verify that the information obtained from the client during the specification was correct. Third, with incremental development, the creation of the software proceeds as a sequence of smaller, more manageable projects. Finally, finishing a version of the software, even an incomplete one, is a satisfying experience that typically generates excitement and increased motivation.

We will discuss the software development process in more detail later in these notes.

After a program (or a version of a program) is specified, it must be designed and implemented. These activities will be discussed in the following sections.

**Study Questions**

1.1.1. What are four properties of a good software specification? *Hint*: Four words that start with the letter *c*.

1.1.2. What is incremental software development?

1.1.3. What are the benefits of incremental development?

1.2 **Design**

We now design the pay calculator. Large programs are always designed as a collection of smaller **software components** so we will proceed in this way (even though the pay calculator is not that large). This **modular** approach has a number of advantages. One is that it makes the software easier to implement because we can focus on one component at a time. Modularity also allows the implementation work to be divided among several programmers. We will discuss modularity in more detail later in this chapter.

Software **design** consists mainly of identifying the components of the software. We may also decide how to store important data or which algorithms
1.2. DESIGN

Pay calculator
Version 1

DESIGN — DRAFT

Components of the program:

- A function that runs the calculator

  Implementation notes: Delegates the execution of the operations on times to separate functions.

- A structure that stores each time as two integers, one for the hours, one for the minutes.

- Functions for operations on times
  - Initializes a time to 99:99.
  - Reads a time from an input stream.
  - Prints a time to an output stream.
  - Computes the difference, in hours, between two times.

Figure 1.3: First draft of the design of the pay calculator

to use to perform major tasks. In the end, we should have a precise specification for each component of the software, together with notes on major implementation details.

Figure 1.3 shows an initial draft of the design of the pay calculator. Several components are identified. The first one is the main function of the program, the one that controls the entire pay calculator. That function will read the input file and need to store times. A structure is a good choice.

The function that runs the calculator will need to perform several operations on times, such as reading a time and computing the difference, in hours, between two times. Instead of performing these operations itself, the function will delegate the execution of these operations to a separate set of functions. This is convenient since it reduces repeated code. But it also separates the details of the time operations from the overall running of the calculator, leading
Global constants:

const float kPayRate = 12;
const char kcsTimesFileName[] = "times.txt";
const char kcsReportFileName[] = "report.txt";

Functions:

- void run_pay_calculator()

  Runs the calculator. See program spec for details.

  Implementation notes: Uses the structure below to store times and delegates the time operations to the associated functions.

Figure 1.4: Design of the pay calculator (part 1 of 2)

to more modularity.

Figures 1.4 and 1.5 show the final design of the pay calculator. It includes three global constants that make the program easier to modify. These can be considered minor components of the program. The file names are stored as C strings.

Each function now has a name, a return value, arguments as well as a precise description of what it does (from the point of view of the user of the function). This is all the information needed to be able to use each function. The design document now fully specifies how the components of the program interact with each other.

Note that we have grouped together the Time structure and the operations that work on times. It makes sense to view this data and these operations as a single component of the program since the only purpose of these operations
1.2. DESIGN

Data types:

- **Time**

  A structure that represents a time as two integers, hours and minutes. Times are on a 24-hour clock (0:00 to 23:59).

Operations:

```cpp
void initialize( Time & t )
Sets t to 99:99.

void read( Time & t, istream & in )
Reads t from in the format h:mm or hh:mm, where each h and m stands for a single digit. No error-checking.

void print( const Time & t, ostream & out )
Prints t to out in the format described for read.

double difference( const Time & t1, const Time & t2 )
Computes the difference, in hours, between t1 and t2. The difference is positive if t1 occurs after t2. (I.e., the difference is computed as "t1 - t2".)
```

Figure 1.5: Design of the pay calculator (part 2 of 2)
is to operate on the Time data.

Note also that the functions print and difference receive their Time arguments by constant reference. This ensures that the arguments cannot be changed accidentally while at the same time avoiding the copying that occurs when arguments are passed by value. If an argument consists of more than one integer, it usually takes less time to pass it by reference (constant or not) than by value.

Study Questions

1.2.1. In what two ways are modular programs easier to implement?

1.2.2. When should an argument be passed by constant reference?

1.3 Names

In the design of the pay calculator, we had to choose several names of variables and functions, as well as one for a structure. The most important consideration is that names should be descriptive. For example, the name of a variable should describe its value. But there is more to it than that. And it’s very important to be consistent. In this section, we describe the naming convention we will follow in these notes.

It is useful to be able to easily distinguish between variable names and names of types. We will do this by using different formats for variables and types. Variable names will use what we will call the underscore format: all lowercase with underscores to separate words. For example, hours and start_time. Names of types, on the other hand, will use the mixed-case format: capitalized words concatenated without any separating character. For example, Time and PhoneBookEntry.

Function names normally occur together with arguments so we will reuse the underscore format for them, as in read and run_pay_calculator.

One exception to these rules concerns compile-time constants. These are constants whose value is determined before the program runs. The pay calculator contains three of those constants: kPayRate, kcsTimesFileName and kcsReportFileName. For compile-time constants, we will use the mixed-case format prefixed by a lowercase k. Note that this does not apply to all constants. For example, the arguments of the function difference are constant but they are not compile-time constants.
1.4. IMPLEMENTATION

When working with a variable, we usually need to know something about its type. Ideally, the type of a variable should be clear from its name so we don’t have to go look for the declaration of the variable. For example, `start_time` is clearly a `Time` and so is `t` when it occurs in the context of a `Time` operation. The constant `kPayRate` is clearly some sort of number and that’s often all we need to know. The name of a variable `kTimesFileName` would clearly indicate that it is some sort of string. But this is likely not to be enough information since C strings are handled very differently than C++ string. This is the reason we included the additional prefix `cs` in the name of the constants `kcsTimesFile` and `kcsReportFile`. For C++ strings, we will use the prefix `s`. For variables, the prefix would be separated from the rest of the name by an underscore as in `cs_input_file_name`.

This use of prefixes to indicate important information about the value of a variable is a variation of what is sometimes called Hungarian notation [Sim]. We will see other examples of prefixes later in these notes.

In general, there are many different naming conventions that can be followed. What is most important is to use some sort of convention and to be consistent. In particular, when working as a team on a project, it is critical for every member of the team to follow the same convention. So if you are asked in an exercise to extend some of the code presented in these notes, you should follow the convention used in these notes.

Study Questions

1.3.1. What format will we use for the names of variables, types and functions?

1.3.2. What are compile-time constants and what format we will use for their names?

1.3.3. What is the meaning of the prefix `cs` in the name `kcsTimesFileName`?

1.4 Implementation

Now that the pay calculator is specified and designed, we can implement it. This means writing and testing the code.

Figure 1.6 shows a possible implementation of the `Time` data type and its operations. Once the data type and its operations are implemented, it is a good idea to test this code right away and on its own. This is called unit


struct Time { int hours, minutes; };

void initialize( Time & t ) { t.hours = t.minutes = 99; }

void read( Time & t, istream & in ) {
    in >> t.hours;
    in.get(); // colon
    in >> t.minutes;
}

void print( const Time & t, ostream & out ) {
    out << t.hours << ':';
    if ( t.minutes < 10 ) out << '0';
    out << t.minutes;
}

double difference( const Time & t1, const Time & t2 ) {
    return (t1.hours + t1.minutes/60.0) - (t2.hours + t2.minutes/60.0);
}

Figure 1.6: The Time data type and its associated functions

testing. Unit testing makes it easier to locate the sources of errors, especially in a large program. In our example, we could test the Time code by using the interactive test driver shown in Figure 1.7. A test driver is a piece of code whose only purpose is to test another component.

Note that testing is critically important. No matter how much care we put into writing correct code, we always make mistakes. For example, in the difference function, notice that we divide the minutes by 60.0 and not 60. This is because the minutes are integers and if we divided by 60, another integer, C++ would perform integer division, meaning that the result would be an integer. In other words, the fractional part of the quotient would be dropped leading 45/60 to evaluate to 0 instead of 0.75, for example. This would have been a subtle bug that may have gone undetected without carefully testing the Time operations on a variety of times.

Figure 1.8 shows the implementation of the run_pay_calculator function. All we now need is a main function that simply calls
int main() {
    Time t1, t2;
    initialize(t1);
    initialize(t2);
    cout << "Initial values: ";
    print( t1, cout );
    cout << '"';
    print( t2, cout );
    cout << " endl;"

    while (true) {
        cout << "Enter two times: ";
        read( t1, cin );
        read( t2, cin );
        cout << "The difference between ";
        print( t1, cout );
        cout << " and ";
        print( t2, cout );
        cout << " is " << difference( t2, t1 ) << " endl;"
    }

    return 0;
}

Figure 1.7: A test driver for Time and its operations
void run_pay_calculator() {
    ifstream ifs_times( kcsTimesFileName );
    ofstream ofs_report( kcsReportFileName );

    int employee_number = -1; // employee number
    Time start_time, stop_time;
    initialize( start_time );
    initialize( stop_time );

    while ( ifs_times >> employee_number ) {
        // read start and stop times
        read( start_time, ifs_times );
        read( stop_time, ifs_times );

        // compute pay
        double pay = difference( stop_time, start_time )
            * kPayRate;

        // print results
        ofs_report << employee_number << ' ';
        print( start_time, ofs_report );
        ofs_report << ' ';
        print( stop_time, ofs_report );
        ofs_report << "$" << fixed << setprecision(2) << pay
            << endl;
    }

    ifs_times.close();
    ofs_report.close();
}

Figure 1.8: The run_pay_calculator function
1.4. IMPLEMENTATION

```c
int main() {
    run_pay_calculator();

    return 0;
}
```

Figure 1.9: The main function

run_pay_calculator, as shown in Figure 1.9.

All the code and documentation of this first version of the pay calculator is available on the course web site under PayCalculator1.

Study Questions

1.4.1. What two activities does the implementation of software involve?

1.4.2. What is unit testing?

1.4.3. What is the main advantages of unit testing?

1.4.4. What is a test driver?

Exercises

1.4.5. Add to the Time data type an operation `is_later_than(t1, t2)` that evaluates to `true` if Time `t1` occurs later than Time `t2`. (Otherwise, the function evaluates to `false`.) **Hint:** Consider carefully how the arguments should be passed to the function.

1.4.6. Add to Time an operation `add_minutes(t, num_minutes)` that adds the number of minutes `num_minutes` to Time `t`. The argument `num_minutes` is an integer. Allow for the number of minutes to be arbitrarily large and even negative. When going forward past 23:59, just cycle back to 0:00. When going backwards past 0:00, cycle forward to 23:59.

1.4.7. Modify the pay calculator as described below. For each part, revise the specification, design and implementation of the program. Modify the original specification, design and implementation of the program as
CHAPTER 1. ABSTRACTION

little as possible but feel free to create new functions to keep the program as modular as possible. Follow the same naming convention already used in the program. For each part of the exercise, which of the program’s components did you modify?

a) Times are on a 12-hour clock, using “a.m.” or “p.m.” Times should be read and printed in that format.

b) Employees are paid $18 per hour for any amount of time worked beyond 8 hours. (Separate the computation of the pay from the function that runs the calculator by creating a new function that computes the pay.)

c) The start and stop times are given in separate files called \texttt{start\_times.txt} and \texttt{stop\_times.txt}. Each line in these files contains an employee number and a time. Employees are listed in the same order in both files.

d) The program reads another file called \texttt{wages.txt} that lists how much each employee should be paid for an hour of work. Each line in this file contains an employee number and an hourly rate. The lines in the wages file are listed in increasing order of employee number. Note that employees do not necessarily work every day. Therefore, some may appear in the wages file but not in the times file (or files). (But you can assume that every employee that appears in the times files also appears in the wages file.) \textit{Hint}: Start with a first version where you assume that every employee works every day.

1.4.8. Create a data type \texttt{Date} that consists of dates such as January 22, 2012. To keep things simple, assume that every month of every year has exactly 30 days. Include the following operations in the data type:

a) Operations \texttt{initialize(date)} and

\begin{verbatim}
initialize(date, month, day, year)
\end{verbatim}

The first one initializes \texttt{date} to January 1, 2000. The second one initializes \texttt{date} to the given month, day and year. The arguments \texttt{month}, \texttt{day} and \texttt{year} are integers.

b) An operation \texttt{read(date, in)} that reads \texttt{date} from input stream \texttt{in}. Dates are typed as \texttt{m/d/y} where \texttt{m}, \texttt{d} and \texttt{y} are integers. No error-checking is performed.
1.4. IMPLEMENTATION

c) A method \texttt{print(date, out)} that prints \texttt{date} to output stream \texttt{out}. Dates are printed in numerical format, as in 1/22/2012.

d) A method \texttt{print.in.words(date, out)} that also prints \texttt{date} to output stream \texttt{out} but with the month in words, as in \texttt{January 22, 2012}.

e) A method \texttt{add(date, num\_days)} that advances \texttt{date} by the number of days \texttt{num\_days}. The argument \texttt{num\_days} is an integer. That number could be arbitrarily large and even negative. \textit{Hint}: This is where the assumption that every month has exactly 30 days is useful.

1.4.9. Create a data type \texttt{ThreeDVector} that consists of three-dimensional vector of real numbers, such as \((3.5, 2.64, -7)\). Include the following operations in the data type:

a) Operations \texttt{initialize(v)} and \texttt{initialize(v, x, y, z)} that initialize \texttt{ThreeDVector v} to \((0, 0, 0)\) and \((x, y, z)\), respectively.

b) An operation \texttt{read(v, in)} that reads \texttt{ThreeDVector v} from input stream \texttt{in}. Vectors are entered in the format \((x, y, z)\) where \(x, y\) and \(z\) are real numbers. No error-checking is performed.

c) An operation \texttt{print(v, out)} that prints \texttt{ThreeDVector v} to output stream \texttt{out}. Vectors are printed in the format described for \texttt{read}.

d) An operation \texttt{add(v1, v2)} that returns the sum of \texttt{ThreeDVector’s v1} and \texttt{v2}.

1.4.10. Create a data type \texttt{Fraction} that consists of fractions, that is, numbers of the form \(a/b\) where \(a\) is an integer and \(b\) is a positive integer. Include the following operations in the data type:

a) Operations

\begin{verbatim}
initialize(f)
initialize(f, a)
initialize(f, a, b)
\end{verbatim}
that initialize the Fraction \( f \) to 0, \( a \) and \( a/b \), respectively. The arguments \( a \) and \( b \) are integers and \( b \) is assumed to be positive. No error-checking is performed.

b) An operation \( \text{read}(f, \text{in}) \) that reads Fraction \( f \) from input stream \( \text{in} \). Fractions are entered in the format \( a/b \) where \( a \) is an integer and \( b \) is a positive integer. No error-checking is performed.

c) An operation \( \text{print}(f, \text{out}) \) that prints Fraction \( f \) to output stream \( \text{out} \). Fractions are printed in the format described for \( \text{read} \). Fractions are not reduced.

d) An operation \( \text{print}_\text{mixed}(f, \text{out}) \) that prints Fraction \( f \) to output stream \( \text{out} \). Fractions are printed in mixed form \( n a/b \) where \( n \) is an integer and \( a/b \) is an optional positive proper fraction (one in which the numerator is less than the denominator). For example, \(-5 6/8\). The fraction is not reduced.

e) Operations \( \text{add}(r, s) \) and \( \text{multiply}(r, s) \) that return, respectively, the sum and product of Fraction’s \( r \) and \( s \). For example, if \( r \) is \( 2/3 \) and \( s \) is \( 3/4 \), then \( \text{add}(r, s) \) returns a Fraction whose value is \( 17/12 \) and \( \text{multiply}(r, s) \) returns \( 6/12 \). The returned fractions are not reduced.

1.5 Modularity and Abstraction

As mentioned earlier, large programs are always designed as collections of smaller software components. In the current version of our pay calculator, all the components are functions but we will see in the next chapter that software components can also correspond to data.

In a well-designed modular program, software components should satisfy the following two properties:

1. **Cohesion**: each component performs one well-defined task.

2. **Independence**: each component is as independent as possible from the others.

In the context of software, independence is achieved when changes to one component do not affect other components. Notice that the definition of modularity says that components should be as independent *as possible* from each
1. Modular programs are easier to understand because their components perform well-defined tasks and can be understood in isolation. In addition, modular programs contain little redundant code.

2. Modularity facilitates software reuse because components perform well-defined tasks that may occur elsewhere and because components depend as little as possible on the context in which they are used.

3. Modular programs are easier to implement because components can be coded one at a time and the work can be divided among various programmers.

4. Modular programs are easier to test because components can be tested in isolation. This simplifies locating and fixing errors.

5. Modular programs are easier to modify because changes to one component often only affect that component.

Figure 1.10: Advantages of modularity

other. Independence is a property that can be achieved at various degrees. The goal is to maximize independence.

Components with a high degree of independence are said to have a low degree of coupling.

Modular programs have a number of advantages. For example, a component that performs one well-defined task is going to be easier to understand and also more likely to be useful in another project. This is not true of a component that performs several unrelated tasks.

Independence, on the other hand, allows us to code, test and modify components in isolation. Independence is also necessary for software reuse: a component that is highly dependent on a particular context will be difficult to use in a different one.

Modularity also typically reduces redundant code because the same component can be reused for repeated tasks. This helps make programs easier to understand and easier to modify. Figure 1.10 summarizes the advantages of modularity.

Independence is usually achieved through information hiding, that is, by having components hide information from each other. Of course, the term
hiding here is somewhat figurative: components aren’t people. We consider that component \( A \) hides information from component \( B \) if component \( B \) was created as if some aspect of component \( A \) was not known.

With functions, a high degree of information hiding and independence is essentially automatic. For example, consider the function \( \text{read} \) of the pay calculator. We know \textit{what} that function does: it reads a time. We also know \textit{how} that function does what it does: those implementation details are shown in Figure 1.6. What is critical to note is that another function, such as \( \text{run} \_{\text{pay\_calculator}} \), depends on \textit{what} \( \text{read} \) does, not on \textit{how} it does it.

For example, the format of the times in the input file could change. (See the exercise of the previous section.) This would require the implementation of \( \text{read} \) to be revised but the other functions of the program would not need to change because they can only depend on what \( \text{read} \) does, not on how it does it.

Independence is usually achieved by having the users of a component depend on its purpose (what the component does) but not on its implementation (how the component does what it does). This is called \textit{abstraction}. In the case of functions, we call it \textit{procedural abstraction}.

Abstraction is one of the most important techniques for the design of modular programs. Abstraction is automatic with functions but it isn’t with other kinds of software components such as data types. With those components, abstraction must be designed into the software. This is why it is important to understand what abstraction is and how to achieve it. Learning to apply abstraction to data is one of the main objectives of these notes and the subject of the next chapter.

Notice that abstraction leads to two clearly separate perspectives on each software component. From the point of view of its users, a component can be seen as having a purpose but no implementation: it is an \textit{abstract} component. This is the \textit{outside} or \textit{public} view of the component. This view concerns only \textit{what} the component does and it includes the \textit{interface} of the component, which is the information needed to communicate with the component, such as the name, return type and arguments of a function.

The developer of a component has a different view: he or she also sees the implementation of the component. This is the \textit{inside} or \textit{private} view. It concerns \textit{how} the component does what it does and it includes the \textit{inner workings} of the component.
Study Questions

1.5.1. What exactly is a modular program?

1.5.2. Give five distinct advantages of modular programs.

1.5.3. Why does eliminating redundant code make programs easier to (a) understand and (b) modify?

1.5.4. What is abstraction?
Chapter 2

Data Abstraction

In this chapter, we will learn how to apply abstraction to data. This will allow us to produce software that is much more modular than what can be achieved by procedural abstraction alone.

2.1 Introduction

The pay calculator we created in the previous chapter is modular. As explained in Section 1.5, in that program, independence between components is achieved through procedural abstraction. For example, the function `run_pay_calculator` depends on what the function `read` does but not on how it does it. As a consequence, if the implementation of `read` changed, there would be no need to revise `run_pay_calculator`.

But now consider the Time data. Each time is currently stored as a pair of integers, one for the hours and one for the minutes (as shown in Figure 1.6). Suppose that, for some reason, we decided to change that. For example, suppose we decided it is better to store each time as a single number of minutes since midnight. Under this representation, for example, 8:30 would be stored as 510. What portions of the pay calculator program would need to be revised?

The definition of `Time` and the implementation of the `Time` operations shown in Figure 1.6 would obviously need to be revised. This is not a surprise since all of this code depends crucially on exactly how the times are stored. But what about the `run_pay_calculator` function?

By examining the implementation of the `run_pay_calculator` function (Figure 1.8), we can see that there is nothing in this code that depends on the
CHAPTER 2. DATA ABSTRACTION

fact that each Time value is a structure consisting of two integers hours and minutes. In particular, run_pay_calculator never directly accesses those integers. Whenever the function needs to do anything to a time, it uses one of the four Time operations initialize, read, print and difference.

This can be illustrated by the component diagram shown in Figure 2.1. In this diagram, an arrow from one component to another means that the first component uses the second. (The name of run_pay_calculator has been abbreviated.) Note that the arrow going from run_pay_calculator to Time does not go inside the box that represents the data type. This reflects the fact that run_pay_calculator never directly accesses the Time data (hours and minutes). In contrast, the arrows going from the operations to Time do reach inside the box.

The fact that run_pay_calculator never directly accesses the Time data has an important consequence: if we changed how times are stored, we would need to revise the definition of the Time data, as well as the implementation of the Time operations, but not the run_pay_calculator function.

All of this points to a general technique for ensuring that functions that use a data type do not depend on the storage details of that data type: instead
of having these functions directly access the data, have them work with the data through a set of operations. Then, if the storage details of the data change, all that needs to be revised is the definition of the data type and the implementation of the operations but not the functions that use the data. This can significantly reduce the amount of work because the number of users of a data type is often much larger than the number of operations of that data type.

Note that if a program is designed in this way, the functions that use the data type depend on what that data is (for example, a time on a 24-hour clock) but not on how that data is stored (with one or two integers, for example). The functions also depend on what the operations do (read, print, etc.) but not on how the operations do what they do. What we have here is abstraction applied to data: data abstraction.

With data abstraction, from the point of view of a user function, a data type looks as if it has a purpose but no implementation: we call this an abstract data type (ADT).

Data abstraction plays a major role in the design of modular programs. In contrast to procedural abstraction, data abstraction is not automatic: it must be designed into the software. In the remainder of this chapter, we will learn concepts and techniques related to the design and implementation of ADT’s.

Study Questions

2.1.1. What is an advantage of treating a data type as an ADT?

2.2 Classes to Enforce Data Abstraction

In the last section, we saw that run_pay_calculator works with Time values by using the Time operations instead of accessing the Time data directly. In other words, run_pay_calculator treats the Time data type as an ADT. But nothing prevents run_pay_calculator from directly accessing the Time data, either by accident or by lack of discipline (on the part of the programmer). In this section, we will learn how to enforce data abstraction by preventing a function such as run_pay_calculator from directly accessing the data of a data type such as Time.

The simplest way of achieving this is to turn Time into a class and declare that the Time data (which consists of the integers hours and minutes) is
private. This prevents users of the class from directly accessing the data. We then declare that the Time operations are friends of the class so they have permission to access the data. The result is shown in Figure 2.2.

In the implementation of the operations, the data members of the class (hours and minutes, in this case) are accessed by using the same notation that is used to access the members of a structure.

Note that classes allow us to do more than simply control access to data, as we will see in the next section.

A version of the pay calculator that uses the class Time of this section can be found on the course web site under PayCalculator2.0.

**Study Questions**

2.2.1. What is the main disadvantage of the Time data type presented in the previous section?

2.2.2. What does it mean for a class to grant friendship to a function?

**Exercises**

2.2.3. Modify the pay calculator as described below. For each part, revise the specification, design and implementation of the program. Modify the original specification, design and implementation of the program as little as possible. For each part of the exercise, which of the program’s components did you modify?

a) Times are stored as a single number of minutes since midnight.

b) Times are read and printed with seconds, in the format h:mm:ss or hh:mm:ss where each h, m and s stands for a single digit.

**2.3 Classes to Support Object-Oriented Programming**

Classes allow us to enforce data abstraction by preventing users from directly accessing the data. But classes have another benefit: they support object-oriented programming (OOP). This section briefly discusses the basic idea and rationale behind object-oriented programming.
class Time
{
    friend void initialize( Time & t );
    friend void read( Time & t, istream & in );
    friend void print( const Time & t, ostream & out );
    friend double difference( const Time & t1, const Time & t2 );

private:
    int hours, minutes;
};

void initialize( Time & t ) { t.hours = t.minutes = 99; }

void read( Time & t, istream & in ) {
    in >> t.hours;
    in.get(); // colon
    in >> t.minutes;
}

void print( const Time & t, ostream & out ) {
    out << t.hours << ':';
    if ( t.minutes < 10 ) out << '0';
    out << t.minutes;
}

double difference( const Time & t1, const Time & t2 ) {
    return (t1.hours + t1.minutes/60.0)
        - (t2.hours + t2.minutes/60.0);
}

Figure 2.2: A Time class with friend operations
Techniques like modularity and abstraction have been developed to help in the construction of large programs. Object-oriented programming is another one of those techniques.

The way most people usually learn to program is called imperative programming. In imperative programming, a program is viewed as a sequence of instructions that tell the computer what to do. Imperative programming works well with small programs but it is not as effective with large programs.

In object-oriented programming, a program is viewed as a collection of objects that work together to accomplish the overall goal of the program. Each object has a certain set of responsibilities. Objects collaborate by requesting services from each other. They do so by sending messages to other objects. The receiver of a message responds by following a predetermined method. The set of messages understood by an object, as well as the methods used to respond to those messages, are determined by the type, or class, of the object.

For example, right now, in the pay calculator, we initialize and read start_time by calling the appropriate functions:

```plaintext
initialize( start_time );
read( start_time, ifs_times );
```

In a sense, start_time just sits there waiting for us to do things to it.

In contrast, in the object-oriented view of programming, start_time is an object with responsibilities. When needed, we ask start_time to initialize and read itself:

```plaintext
start_time.initialize();
start_time.read( ifs_times );
```

On receiving the initialize and read messages, start_time responds by executing the corresponding method (which would be defined in the class Time). Note that messages, like read above, can have arguments.

Here’s another example. To compute the difference between start_time and stop_time, we currently call a function on both times:

```plaintext
difference( stop_time, start_time )
```

The object-oriented way is to ask one of the times to tell us the difference between itself and the other time:

```plaintext
stop_time.minus( start_time )
```
2.3. **CLASSES TO SUPPORT OOP**

```cpp
class Time {
  public:
    void initialize() { hours = minutes = 99; }
    void read( istream & in ) {
      in >> hours;
      in.get(); // colon
      in >> minutes;
    }
    void print( ostream & out ) const {
      out << hours << ':';
      if ( minutes < 10 ) out << '0';
      out << minutes;
    }
    double minus( const Time & t2 ) {
      return (hours + minutes/60.0)
        - (t2.hours + t2.minutes/60.0);
    }
  private:
    int hours, minutes;
};
```

Figure 2.3: A class Time with methods

Note that in this last example, the Time objects play different roles: one is the receiver while the other is an argument. (In addition, we changed the name of the operation from difference to minus because t1.minus(t2) reads better than t1.difference(t2).)

Figure 2.3 shows the class Time with all the time operations turned into methods. The methods are declared public so they can be accessed by the users of Time. (We will see examples of private methods later.) Note that methods are sometimes called member functions.

In the implementation of the methods, the data members of the receiver are accessed without specifying an object. For example, in the implementation
of minus, the hours of the receiver are accessed as hours while the hours of t2 are accessed as t2.hours.

One way to make sense of this (and to remember how it works), is to read a method from the perspective of the receiver, as if the method was telling the receiver how to respond to the message. So the minus method is telling the receiver to add its hours to its minutes divided by 60, and to subtract from that t2’s hours and t2’s minutes divided by 60.

Figure 2.4 shows a revised run_pay_calculator function that uses the new class Time. (Compare with Figure 1.8.) The new design of the program is illustrated by the component diagram shown in Figure 2.5.

The main advantage of OOP is two-fold. First, it automatically produces a lot of data abstraction, which results in a high degree of independence between components. Second, OOP encourages software components to delegate as many tasks as possible to other components, just as we would when organizing the members of a group of people in the real world. This leads to a high degree of cohesion in components, which is the other key aspect of modularity.

Note that in pure OOP, every component of a program is an object. With languages such as C++ and Java, we normally use a mix of object-oriented and imperative programming.

A version of the pay calculator that uses the class Time of this section can be found on the course web site under PayCalculator2.1.

**Study Questions**

2.3.1. In OOP, how is an object viewed differently from just a piece of data?

2.3.2. What is a message? What is a method? What is a receiver?

2.3.3. What is the ultimate goal of both data abstraction and OOP?

**Exercises**

2.3.4. Turn the operation is_later_than(t1, t2) of Exercise 1.4.5 into a method. In other words, add to the class Time of this section a method is_later_than(t2) that evaluates to true if the receiver occurs later than the Time argument t2.

2.3.5. Turn the operation add_minutes(t, num_minutes) of Exercise 1.4.6 into a method. In other words, add to the class Time of this section a
void run_pay_calculator() {
    ifstream ifs_times( kcsTimesFileName );
    ofstream ofs_report( kcsReportFileName );

    int employee_number = -1; // employee number
    Time start_time, stop_time;
    start_time.initialize();
    stop_time.initialize();

    while ( ifs_times >> employee_number ) {
        // read start and stop times
        start_time.read( ifs_times );
        stop_time.read( ifs_times );

        // compute pay
        double pay = stop_time.minus( start_time ) * kPayRate;

        // print results
        ofs_report << employee_number << ' ';
        start_time.print( ofs_report );
        ofs_report << ' ';
        stop_time.print( ofs_report );
        ofs_report << " $" << fixed << setprecision(2) << pay
                    << endl;
    }

    ifs_times.close();
    ofs_report.close();
}

Figure 2.4: A revised run_pay_calculator function
method add_minutes(num_minutes) that adds the number of minutes num_minutes to the receiver.

2.3.6. Turn the data type Date of Exercise 1.4.8 into a class. In other words, each object in this class represents a date such as January 22, 2012. To keep things simple, assume that every month of every year has exactly 30 days. Include the following methods in the class:

a) Methods initialize() and initialize(month, day, year). The first one initializes the receiver to January 1, 2000. The second one initializes the receiver to the given month, day and year. The arguments month, day and year are integers.

b) A method read(in) that reads the receiver from input stream in. Dates are typed as m/d/y where m, d and y are integers. No error-checking is performed.

c) A method print(out) that prints the receiver to output stream out. Dates are printed in numerical format, as in 1/22/2012.

d) A method print_in_words(out) that also prints the receiver to output stream out but with the month in words, as in January 22, 2012.

e) A method add(num_days) that advances the receiver by the number of days num_days. The argument num_days is an integer. That number could be arbitrarily large and even negative.
2.3. CLASSES TO SUPPORT OOP

2.3.7. Turn the data type ThreeDVector of Exercise 1.4.9 into a class. Each object in this class represents a three-dimensional vector of real numbers, such as \((3.5, 2.64, -7)\). Include the following methods in the class:

a) Methods `initialize()` and `initialize(x, y, z)` that initialize the receiver to \((0, 0, 0)\) and \((x, y, z)\), respectively.

b) A method `read(in)` that reads the receiver from input stream `in`. Vectors are entered in the format \((x, y, z)\) where `x`, `y` and `z` are real numbers. No error-checking is performed.

c) A method `print(out)` that prints the receiver to output stream `out`. Vectors are printed in the format described for `read`.

d) A method `add(v2)` that returns the sum of the receiver and ThreeDVector `v2`.

2.3.8. Turn the data type `Fraction` of Exercise 1.4.10 into a class. Each object in this class represents a number of the form \(a/b\) where `a` is an integer and `b` is a positive integer. Include the following methods in the class:

a) Methods

```python
initialize()
initialize(a)
initialize(a, b)
```

that initialize the receiver to 0, `a` and `a/b`, respectively. The arguments `a` and `b` are integers and `b` is assumed to be positive. No error-checking is performed.

b) A method `read(in)` that reads the receiver from input stream `in`. Fractions are entered in the format \(a/b\) where `a` is an integer and `b` is a positive integer. No error-checking is performed.

c) A method `print(out)` that prints the receiver to output stream `out`. Fractions are printed in the format described for `read`. Fractions are not reduced.

d) A method `printMixed(out)` that prints the receiver to output stream `out`. Fractions are printed in mixed form \(n \ a/b\) where `n` is an integer and `a/b` is an optional positive proper fraction (one in


```cpp
void println( const Time & t, ostream & out ) {
    t.print(out);
    out << endl;
}
```

Figure 2.6: The println function

which the numerator is less than the denominator). For example, $-5 \ 6/8$. The fraction is not reduced.

e) Methods add(s) and multiply(s) that return, respectively, the sum and product of the receiver and Fraction s. For example, if the receiver $r$ is $2/3$ and $s$ is $3/4$, then $r.add(s)$ returns a Fraction whose value is $17/12$ and $r.multiply(s)$ returns $6/12$. The returned fractions are not reduced.

2.4 Constant Methods

We now have the basic tools needed for implementing ADT’s and for programming in an object-oriented way. In particular, we know about classes, privacy, methods and friendship. In the remaining sections of this chapter, we will refine these tools. In particular, we will consider several improvements to our class Time.

We begin in this section by addressing a major flaw of Time. Consider the function shown in Figure 2.6. This function prints a time and then moves to the next line. The code looks good but it won’t compile because the Time argument of println is declared constant and the compiler will complain that the method print may attempt to change its receiver.

The solution is to declare the method print to be a **constant method**. This is done by adding the keyword `const` right after the argument list of the method:

```cpp
void print( ostream & out ) const
```

This essentially declares that the receiver of the print method is constant and cannot be changed by the method.

This has two consequences. First, the compiler will not allow print to change its receiver. Second, the compiler will allow print to be used on a
constant `Time`. In general, a constant method is not allowed to change its receiver and only messages corresponding to constant methods can be sent to constant objects.

Variables should be declared constant whenever possible because this helps to prevent errors. As a consequence, methods should be declared constant whenever possible so they can be used on constant objects. For example, in our class `Time`, `minus` should also be declared constant:

```cpp
double minus(const Time & t2) const
```

Source code and documentation for a revised version of the class `Time` is available on the course web site under `Time1.0`.

### Study Questions

2.4.1. How do you prevent a method from modifying its receiver?

### Exercises

2.4.2. Which of the new `Time` methods described in the exercises of the previous section should be declared constant?

2.4.3. Consider the new classes described in the exercises of the previous section. In each of these classes, which methods should be constant?

### 2.5 Inline Methods

The methods of our class `Time` are currently declared and implemented `inside` the class declaration, as shown in Figure 2.3. This has the effect of making them `inline` methods.

Compilers treat inline functions differently from ordinary functions. When a compiler sees a call to an inline function, it removes the function call and replaces it by the body of the function. In principle, this should speed up the program because it avoids the extra work involved in calling a function. But it may also increase the size of the program and very large programs can run more slowly because they can’t be stored entirely within the fastest portions of a computer’s memory.

The only sure way to know if it’s better to make a function inline is to run tests. But this is often impractical so several rules of thumb have been
inline void println( const Time & t, ostream & out ) {
    t.print(out);
    out << endl;
}

Figure 2.7: An inline standalone function

proposed. One is to make a function inline if it consists of no more than ten lines of code and includes no loops or `switch` statements.

As mentioned above, methods can be made inline by declaring and implementing them within the class declaration. A standalone function (a function that is not a method of any class) can be made inline by preceding its implementation with the keyword `inline`, as shown in Figure 2.7.

Inline methods can make a class declaration become crowded. The preferred style in C++ is for a class declaration to be mainly a list of methods and variable declarations. Long methods are usually implemented outside of the class declaration. This is illustrated in Figure 2.8 where methods longer than a single statement are now implemented outside the class declaration. Note that these methods are still inline because their implementations are preceded by the keyword `inline`. Note also that the name of each method that’s implemented outside the class declaration is preceded by the name of the class. This tells the compiler that these functions are methods that belong to the class `Time` and not standalone functions.

Source code for the revised class `Time` of Figure 2.8 is available on the course web site under `Time1.1`.

**Study Questions**

2.5.1. What is special about an inline function?

2.5.2. What are two ways to make a method inline?

**Exercises**

2.5.3. Revise the classes in the exercises of the previous sections by moving out of the class declaration the implementation of all the methods that are longer than a single statement. But make sure short methods stay inline. Use the rule of thumb given in this section.
class Time
{
public:
    void initialize() { hours = minutes = 99; }

    void read( istream & in);
    void print( ostream & out ) const;

    double minus( const Time & t2 ) const {
        return (hours + minutes/60.0)
               - (t2.hours + t2.minutes/60.0);
    }

private:
    int hours, minutes;
};

inline void Time::read( istream & in ) {
    in >> hours;
    in.get(); // colon
    in >> minutes;
}

inline void Time::print( ostream & out ) const {
    out << hours << ':';
    if ( minutes < 10 ) out << '0';
    out << minutes;
}

Figure 2.8: The class Time with some methods implemented outside the class declaration
2.6 Constructors

A common programming mistake is to create a variable but to forget to set it. This error can be difficult to detect because, during testing, the variable may get a random initial value that happens to be the right value, or something close to it. But later, when the program is used in a real-life situation, the error could manifest itself, with possibly catastrophic consequences.

One way to avoid this problem is to get into the habit of always initializing a variable as soon as it is created. For example, if a counter $i$ is needed for a loop, it is better to set it to its initial value right away, as in

```c
int i = 0;
```

This ensures that we don’t forget to set the counter later.

Another example is found in the pay calculator. In the `run_pay_calculator` function (see Figure 2.4), we set `employee_number` as soon as it is created:

```c
int employee_number = -1;
```

Note that in this case, there is no obvious initial value for this variable. So we set it to a clearly invalid value, which should make it easier to detect if ever we later start using the variable without setting it to an appropriate value.

Initialization is also an issue with objects. For example, in the pay calculator, right after we create `start_time`, we initialize it to 99:99 (a clearly invalid value) by using the `initialize` method:

```c
start_time.initialize();
```

One weakness of the above is that it is still possible to omit initializing a variable, either by accident or by laziness. A more robust alternative would be for variables to be initialized automatically. In C++, this is possible in the case of objects.

Every time an object is created, it is always automatically initialized by special method called a constructor. For example, Figure 2.9 shows the class `Time` with three constructors added. Constructors have the same name as the class but they have no return value (not even `void`). Constructors can have arguments.

The constructor with no arguments is called the default constructor. A declaration such as
class Time
{
public:
    Time() { hours = minutes = 99; }
    Time(int h) { hours = h; minutes = 0; }
    Time(int h, int m) { hours = h; minutes = m; }

    void read( istream & in );
    void print( ostream & out ) const;

    double minus( const Time & t2 ) const {
        return (hours + minutes/60.0)
                - (t2.hours + t2.minutes/60.0);
    }

private:
    int hours, minutes;
};
uses the default constructor. The default constructor of class `Time` simply does what the `initialize` method used to do: it initializes the time to the clearly invalid value 99:99. The object produced by the default constructor can be called the default object of the class.

The second and third constructors can be used to initialize `Time` objects to particular values. The second constructor is used in a declaration such as

```
Time noon(12);
```

where it initializes the time to 12:00. The third constructor is used in a declaration such as

```
Time wake_up_time(6,15);
```

where it initializes the time to 6:15. Notice how the arguments of these constructors are provided as part of the declaration of the times.

Constructors can also be called directly to create and initialize objects. For example, suppose that we want to change `wake_up_time` to 6:30. We can do this as follows:

```
Time six_thirty(6,30);
wake_up_time = six_thirty;
```

But we can also do this more concisely:

```
wake_up_time = Time(6,30);
```

This creates a temporary `Time` object, initializes it to 6:30 by using the third constructor, and then copies the new `Time` object to `wake_up_time`.

The second constructor can be used in the same way. For example,

```
wake_up_time = Time(6);
```

will set `wake_up_time` to 6:00.

Note that in this last example, the second constructor is being used essentially to convert the integer 6 into the `Time` object 6:00. We can achieve the same effect by simply writing

```
wake_up_time = 6;
```
2.6. CONSTRUCTORS

What happens here is that the compiler is not able to find an assignment operator that can copy an integer to a Time. But it finds one that copies a Time to a Time, so it looks for a way to convert an integer into a Time. The second constructor of the class provides a way to perform this conversion. This called an **implicit conversion** because it was not explicitly requested by the programmer.

Implicit conversions are performed whenever the compiler expects an object, some other type of value is provided and the class of the object contains a one-argument constructor that can perform the conversion (by taking that other type of value as argument).

Note that a Time variable can be declared and initialized by the second constructor with either the notation

\[
\text{Time noon(12);}
\]

or

\[
\text{Time noon = 12;}
\]

This equal sign is not an assignment operator but just another way of invoking the one-argument constructor.

If you do not include any constructor in a class, then the compiler will automatically generate a default constructor. This compiler-generated default constructor typically does nothing and is usually not what is needed. If you add any constructor to a class but not a default constructor, then the compiler will not generate a default constructor and the class will be left without one. In that case, you would not be able to create arrays of objects of that class since objects in an array are automatically initialized by the default constructor. Therefore, you usually need to include a default constructor in all your classes.

The second and third constructors of our class Time are very similar. In fact, the second constructor can be viewed as a special case of the third constructor, one in which the value 0 is used as the second argument. In C++, it is possible to merge the two constructors into a single constructor as follows:

\[
\text{Time( int h, int m = 0 ) \{} \\
\text{   hours = h;} \\
\text{   minutes = m;} \\
\text{\}}
\]
The constructor specifies that the value 0 is to be used in case the second argument is missing. This default value is usually called a *default argument*.

Default values can only be specified for the rightmost arguments of a function. When the function is called, the values that are given as arguments are matched to the arguments of the function from left to right. Default values are then used in place of the missing arguments. A function can specify default values for all its arguments.

The version of `Time` with constructors is available on the course web site under `Time1.2`.

**Study Questions**

2.6.1. Why is it good practice to initialize a variable as soon as it is declared?

2.6.2. What is a constructor?

2.6.3. What is a default constructor?

2.6.4. When does the compiler perform an implicit conversion?

2.6.5. What type of constructor is used to perform implicit conversions?

2.6.6. When precisely does the compiler automatically generate a default constructor for a class?

**Exercises**

2.6.7. Revise the classes of the exercises of Section 2.3 by turning their `initialize` methods into constructors. Use default arguments as appropriate. (Make sure your use of default arguments doesn’t inadvertently create constructors that were not specified.)

**2.7 Get and Set Methods**

Our class `Time` provides all the functionality we need for the pay calculator. But the range of operations that can be performed on these times is fairly limited.

For example, imagine that later, perhaps in another program, we needed to print times with an `h` instead of a colon, as in `8h30`. (This is how it’s
2.7. **GET AND SET METHODS**

```cpp
class Time {
public:
...

    int hours() const { return hours_; }
    int minutes() const { return minutes_; }

    void set_hours(int new_hours) { hours_ = new_hours; }
    void set_minutes(int new_minutes) {
        minutes_ = new_minutes;
    }
}

private:
    int hours_, minutes_;
```

Figure 2.10: Get and set functions for the class Time

```cpp
void print_with_h(const Time & t) {
    cout << t.hours() << 'h';
    if (t.minutes() < 10) cout << '0';
    cout << t.minutes();
}
```

Figure 2.11: The function print_with_h

done in French, for example.) One option would be to modify the class to include a new print_with_h method. But this requires “reopening” that class: learning again how it works and running the risk of breaking the parts of it that already work. (This wouldn’t be much of an issue with a simple class like Time, but classes can be much larger and much more complex.)

Another option is for the designers of the original class to include so-called *get* and *set* methods in the class, as shown in Figure 2.10. Then, without modifying the class, we can write a print_with_h function that prints times with an *h* instead of a colon, as shown in in Figure 2.11.

Note that we changed the names of the data members of the class from hours and minutes to hours_ and minutes_. This is because in a class, a data member and a method with no arguments cannot share the same name.
Adding an underscore to the names of all the data members allows those names to be used for get methods. It also makes it easy to recognize those data members within the implementation of the methods. We will follow this convention from now on.

Get and set methods are a standard way of adding flexibility to a class because they allow users to perform operations that weren’t anticipated by the designers of the class. This increases the chances that the class will be useful in other projects, which is good.

On the negative side, get and set methods increase dependence between a class and its users. For example, the `run_pay_calculator` function uses the `print` method to print times as in

```java
start_time.print( ofs_report );
```

If ever times were changed to include seconds, `run_pay_calculator` would not need to be modified.

However, now that `Time` has get and set methods, nothing prevents a function from printing times itself, by using the get methods as shown in Figure 2.11.

There are at least two disadvantages to using the get methods instead of the method `print`. First, it requires writing more code. Second, and more importantly, it creates more dependence between `Time` and the user function. For example, the addition of seconds to times would require the user function to be revised.

Therefore, it is better for users of a class not to use the get and set methods to perform a task that can be entirely performed by a method already included in the class. This is an example of the design strategy that software components should delegate as much work as possible to other components. This usually leads to greater independence between software components.

You may be worried that get and set methods reveal too much about the implementation of the class. For example, in the case of `Time`, if we’re going to include get and set methods for hours and minutes, then why not just make the data members `hours` and `minutes` public? But note that users of `Time` already know that times consist of hours and minutes. That’s part of what times are. But that’s not necessarily how times are stored. For example, we could store times as a single number of minutes since midnight. That wouldn’t change the fact that times consist of hours and minutes. Even if `Time` was still implemented in this way, we could still include the get and set methods for
hours and minutes (even though the implementation of those methods would be a little more complicated).

The version of the class Time with get and set methods, as well as a test driver that includes the print with h function, is available on the course web site under Time1.3.

**Study Questions**

2.7.1. Why is it a bad idea for a function to use the get methods instead of the print method to print a time?

**Exercises**

2.7.2. Without modifying the class Time, write a function that takes a Time object as argument and prints the time in the 12-hour format using “a.m.” and “p.m.” This function should be a stand-alone function, not a method of the class Time. Do not use friendship declarations.

2.7.3. Modify the implementation of the class Time so that times are stored as a single number of minutes. Do this without changing the interface of the class so that users of Time do not need to be revised. (In particular, the class should continue to have get and set methods for hours and minutes.)

2.7.4. Add to the class Date of Exercise 2.3.6 a single method set(month, day, year) that sets the date to the given month, day and year. The arguments are integers.

2.7.5. Add the following methods to the class ThreeDVector of Exercise 2.3.7:

   a) A method set(x, y, z) that sets the vector to (x, y, z).
   b) Methods x(), y() and z() that return, respectively, the first, second and third components of the vector.

2.7.6. Add the following methods to the class Fraction of Exercise 2.3.8:

   a) A method set(a, b) that sets the fraction to a/b. The arguments are integers. Assume that b is positive.
   b) Methods numerator() and denominator() that return, respectively, the numerator and denominator of the fraction.
**2.8 Operators**

In the pay calculator, we use the `minus` method of class `Time` to compute the difference between two times as follows:

\[
\text{stop\_time.minus( start\_time )}
\]

But the following code is more natural:

\[
\text{stop\_time} - \text{start\_time}
\]

And because it is more natural, it is easier to remember, easier to write and easier to understand. The essential difference is that the more natural code uses an `operator` instead of a `method`.

In C++, we can extend existing operators so they work with new argument types. This done by creating a new version of the operator. It’s called **operator overloading** because it adds new meaning to an existing operator.

To overload the subtraction operator for times, we simply need to change the name of the method from `minus` to `operator−`, as shown in Figure 2.12. When the compiler sees an expression such as

\[
\text{stop\_time} - \text{start\_time}
\]

it will interpret it as

\[
\text{stop\_time.operator\- ( start\_time )}
\]

and the desired result will be achieved.

As explained earlier, the compiler can use the one-argument constructor of class `Time` to perform implicit conversions from integers to times. So, for example, the code `t - 8` would be perfectly valid and result in the time 8:00 being subtracted from `t`.

```cpp
double operator-( const Time & t2 ) const {
    return (hours_ + minutes_/60.0)
        - (t2.hours_ + t2.minutes_/60.0);
}
```

Figure 2.12: A subtraction operator for `Time`


```c
inline double operator-( const Time & t1, const Time & t2 ) {
    return (t1.hours() + t1.minutes()/60.0)
            - (t2.hours() + t2.minutes()/60.0);
}
```

Figure 2.13: The subtraction operator as a standalone function

On the other hand, it is not possible to write \( 12 - t \) since the left operand of the operator is its receiver, not an argument. Implicit conversions are not performed on receivers.

To get implicit conversions on both operands of the subtraction operator, we must redesign the operator so that both operands are arguments. This can be done by taking the operator out of the class and turning it into a standalone function with two arguments, as shown in Figure 2.13. This is possible because an expression such as

\[
\text{stop time} - \text{start time}
\]

can also be interpreted by the compiler as

\[
\text{operator-}( \text{stop time}, \text{start time} )
\]

Note that if our class didn’t have get methods, we would simply make the operator a friend of the class so we could directly access the private data members `hours_` and `minutes_`.

We can also overload the input (or stream extraction) operator `>>` so that instead of writing

\[
\text{start time}.\text{read( ifs_report )}
\]

we can write

\[
\text{ifs_report} \gg \text{start time}
\]

Just as we do with integers, characters and strings.

The overloading of the input operator involves a couple of particular issues. One has to do with how the compiler interprets code such as

\[
in \gg t
\]

Just as in the case of the subtraction operator, there are two possibilities:
inline istream & operator>>( istream & in, Time & t ) {
    in >> t.hours_;
    in.get(); // colon
    in >> t.minutes_;
    return in;
}

Figure 2.14: An input operator for Time

1. in.operator>>(t)
2. operator>>(in, t)

The first interpretation corresponds to overloading the operator by adding
the method operator>> to the class istream, which is the class of the stan-
dard input stream cin as well as all input file streams. But istream is a
library class and we cannot modify it.

Now, if the compiler could interpret in >> t as
t.operator>>(in)
we would be all set: we could overload the operator by adding the method
operator>> to our class Time. But the C++ compiler does not interpret
in >> t in this way.

So we are left with the second interpretation, which corresponds to over-
loading the operator by creating a standalone function with two arguments,
as shown in Figure 2.14.

Note that here we decided to make the operator a friend of the class and
directly accessing the data members. The alternative would have been to use
the set methods as follows:

    int new_hours = 0;
    in >> new_hours;
    t.set_hours(new_hours);

This would have been less efficient.

1Strictly speaking, input file streams are of class ifstream. We will say more about this
in the next chapter when we take a look at I/O stream classes.
Note that the input operator returns the stream it receives as argument. There are two reasons for this.

First, that return value indicates whether the input operation succeeded: when used in a conditional statement, a stream variable evaluates to `true` if the last operation succeeded and to `false` if the last operation failed. For example, the following loop reads all the times contained in the file associated with the stream variable `f`:

```cpp
while ( f >> t ) { cout << t << endl; }
```

This can be read as follows:

While `t` can be read from `f`, print `t` to the screen.

What actually happens is that as long as the reading operation succeeds in reading a time from the file, the expression `f >> t` evaluates to `true` and the loop continues. But after the last time is read from the file, the next reading operation fails, `f >> t` evaluates to `false` and the loop terminates.

The second reason why the input operator returns the stream has to do with chains of reading operations. In C++, instead of having to write

```cpp
in >> t1;
in >> t2;
```

it is normally possible to write the more convenient and more readable

```cpp
in >> t1 >> t2;
```

Such a statement is actually executed as a sequence of nested function calls:

```cpp
operator>>( operator>>( in, t1), t2);
```

For this to work, the first call to the input operator must return the stream so it becomes the first argument of the second call to the operator.

Note that the operator must return a `reference` to the stream, not a `copy` of the stream. This allows the calling function to access the original stream instead of copy of the stream. This is done for the same reason that stream arguments are passed by reference as arguments: all I/O operations to a single file should normally be performed through a single stream variable, not various copies of it. We will see other examples of functions that return references later in these notes.
 inline ostream & operator<<( ostream & out, const Time & t ) {
    out << t.hours() << ":";
    if ( t.minutes() < 10 ) out << '0';
    out << t.minutes();
    return out;
}

Figure 2.15: An output operator for Time

The output (or stream insertion) operator << can be overloaded in a way similar to the input operator, as shown in Figure 2.15.

Figure 2.16 shows a revision of the run_pay_calculator function that uses the latest version of our class Time. By comparing with the previous version (see Figure 2.4), we see that the default constructor and operators of the class allow us to write more natural code, that is, code that reads better and is easier to both write and understand.

The latest version of the class Time is available on the course website under Time1.4. A version of the pay calculator that uses this class is available under PayCalculator2.2.

Study Questions

2.8.1. What is operator overloading?

2.8.2. What is the advantage of declaring an operator such as the subtraction operator as a stand-alone function instead of a method?

2.8.3. Why do the input and output operators have to be declared as stand-alone functions instead of methods?

2.8.4. Why do the input and output operators return the stream?

Exercises

2.8.5. Transform the is_later_than method that you wrote for Exercise 2.3.4 into the less than operator <. Is it better to define it inside or outside the class?

2.8.6. Add an equality testing operator == to the class Time.
void run_pay_calculator() {
    ifstream ifs_times( kcsTimesFileName );
    ofstream ofs_report( kcsReportFileName );

    int employee_number = -1; // employee number
    Time start_time, stop_time;

    while ( ifs_times >> employee_number ) {
        // read start and stop times
        ifs_times >> start_time >> stop_time;

        // compute pay
        double pay = ( stop_time - start_time ) * kPayRate;

        // print results
        ofs_report << employee_number << ' ' << start_time
        << ' ' << stop_time << " $" << fixed
        << setprecision(2) << pay << endl;
    }

    ifs_times.close();
    ofs_report.close();
}

Figure 2.16: A revised run_pay_calculator function
2.8.7. Add the operators $<<$, $>>$ and $+=\,$ to the class \texttt{Date} of Exercise 2.3.6. They should behave just like the methods read, print and add, respectively.

2.8.8. Add the operators $<<$, $>>$ and $+$ to the class \texttt{ThreeDVector} of Exercise 2.3.7. They should behave just like the methods read, print and add, respectively.

2.8.9. Add the operators $+$ and $\ast$ to the class \texttt{Fraction} of Exercise 2.3.8. They should behave just like the methods plus and times, respectively.

2.8.10. Add the operators $<$ and $==$ to the class \texttt{Fraction} of Exercise 2.3.8. Two fractions such as $2/3\,$ and $8/12\,$ should be considered equal.

2.9 Compiling Large Programs

All the source code of our pay calculator program is currently contained in a single file. This implies that to work on a component in isolation, we have to copy its code out of the program and then back into it. This is inconvenient and prone to errors.

Having all the code in a single file also implies that the entire program needs to be recompiled every time any part of the code is changed. Since compiling a large program can take a significant amount of time, this is also inconvenient, especially during debugging when often only small changes are made to the program between recompilations.

An alternative is to organize the program so that each component is contained in its own file (or files) and then to recompile only those files that need to be recompiled.

The standard way of doing this is to separate each component into a \textit{header file} and an \textit{implementation file}. The header file contains the declarations associated with the component, that is, all the code that the compiler needs to be able to compile code that uses the component. The implementation file contains the rest of the code. The names of these files normally end with the extensions \texttt{h} and \texttt{cpp}, respectively.

For example, we can reorganize the pay calculator into five files, as follows:

1. \texttt{Time.h}: the declaration of the class \texttt{Time} and the implementation of all the inline \texttt{Time} operations (whether methods or standalone functions), as shown in Figure 2.17.


```cpp
#ifndef _Time_h_
#define _Time_h_

#include <iostream>

class Time
{
  public:
    friend std::istream & operator>>( std::istream & in, Time & t );

    Time() { hours_ = minutes_ = 99; }
    Time( int h, int m = 0 ) { hours_ = h; minutes_ = m; }

    int hours() const { return hours_; }
    int minutes() const { return minutes_; }

    void set_hours( int new_hours ) { hours_ = new_hours; }
    void set_minutes( int new_minutes ) {
      minutes_ = new_minutes;
    }

  private:
    int hours_, minutes_; 

};

... // implementation of the inline operations

#endif
```

Figure 2.17: The header file `Time.h`
```c
#ifndef _run_pay_calculator.h_
#define _run_pay_calculator.h_

// Global constants
const float kPayRate = 12;
const char kcsTimesFileName[] = "times.txt";
const char kcsReportFileName[] = "report.txt";

void run_pay_calculator();
#endif
```

Figure 2.18: The header file `run_pay_calculator.h`

2. `Time.cpp`: the implementation of any `Time` operation not already implemented in the header file. (There are none in our case.)

3. `run_pay_calculator.h`: the declarations of the global constants and of the `run_pay_calculator` function, as shown in Figure 2.18.


5. `main.cpp`: the main function.

Note that the `main` function does not need to be split into header and implementation files because that function will not be used by any other component.

Each implementation file should include \( \#include \) the corresponding header file, as well as library files and other header files that are needed. For example, `run_pay_calculator.cpp` includes `run_pay_calculator.h`, `fstream`, `iostream`, `iomanip` and `Time.h`.

Each header file should include all the necessary library and header files. In the pay calculator, `payCalculator.h` does not need to include anything (see Figure 2.18). But `Time.h` includes `iostream` (see Figure 2.17).

Note that no global `using` declarations are used in `Time.h`. Instead, the long form `std::istream` is used to access `istream`. This is because header files such as `Time.h` will be included in the source code of other
2.9. COMPILING LARGE PROGRAMS

components and it’s best to avoid “polluting” the namespaces of those components. Therefore, in a header file, all using declarations should be local to individual functions. This is not an issue with implementation files, which is why run_pay_calculator.cpp contains declarations such as
using std::ifstream and using std::endl.

With a compiler that is part of an integrated development environment (IDE), we normally create a project, add to it all the source files and then just click a button to have the IDE compile the project. When recompiling a program, the IDE will automatically compile only the files that actually need to be compiled. As mentioned before, with a large program, this can take much less time than recompiling the entire program.

With a console-type compiler such as g++ for Unix and Linux, the entire program can be compiled by compiling all the cpp files as follows:

g++ *.cpp

Afterwards, the process of recompiling only those files that are needed is usually managed with the assistance of the make utility. This will be discussed briefly in the next section.

Now suppose that another component is added to the pay calculator and that this new component uses both the pay calculator functions and the class Time. The header file of that new component would normally include both payCalculator.h and Time.h. But this would not work if ever Time.h was already included in payCalculator.h. In that case, the compiler would complain that the class Time is being declared twice.

A solution is to remember which header files are already included in which other header files. In a large program, this would be difficult to manage and prone to errors.

A simpler solution is to use the trick shown in Figures 2.17 and 2.18. At the beginning of Time.h, for example, we test to see if the symbol _Time_h_ is defined. If not, we define the symbol and proceed with the rest of the file. The second time the file is visited, the symbol will be defined, the test will fail and the contents of the file will be skipped all the way to the endif directive at the very bottom. This prevents the contents of the file from being read more than once.

It is a good idea to use this trick systematically in all header files. Each header file must use a unique symbol name. A scheme based on a modification of the file name works well. For example, for file named component.h, use the symbol _component.h_.

A version of the pay calculator organized with header and implementation files for separate compilation is available on the course web site under PayCalculator2.3.

### Study Questions

2.9.1. Why is the setup described in this section especially useful during debugging?

2.9.2. Why is it better not to use global using declarations in a header file?

### Exercises

2.9.3. By following the guidelines given in this section, reorganize the code of the classes you created for the exercises of this chapter.

### 2.10 The make Utility

With a console-type compiler such as g++, it is possible to manually recompile only the files that need to be recompiled. The first time we compile the program, we would compile all the implementation files as follows:

```sh
  g++ -c *.cpp
  g++ *.o -o paycalc
```

The first command compiles the implementation files and produces object files with the .o extension. The second command links those files to produce the executable paycalc. Afterwards, if, for example, only the file Time.cpp is changed, then we only need to do the following:

```sh
  g++ -c Time.cpp
  g++ *.o -o paycalc
```
The difficulty with this manual approach is that if we modify a header file, then we need to figure out which files include it, either directly or indirectly, because all those files will need to be recompiled. With a large program, this is obviously not practical.

Fortunately, the process can be automated by using the `make` utility. All the commands needed for the compilation of the program are put in a `Makefile` along with a declaration of dependencies between the various source files, as shown in Figure 2.19. For example, the entry

```
main.o: main.cpp run_pay_calculator.h
    g++ -c main.cpp -o main.o
```

indicates that the file `main.o` should be generated using the command

```
g++ -c main.cpp -o main.o
```

whenever any of the other three files changes. This list of dependencies contains all the files that are included, directly or indirectly, in `main.cpp`. The list of dependencies should normally be on a single line and the command line that follows should begin with a TAB character.
To compile the program using the Makefile, simply type make or make paycalc. To run the cleaning command included in the Makefile, type make clean. This will delete the executable paycalc as well as the object files and all files with a name that ends in a tilde (~). This includes the backup files produced by the emacs text editor.

The Makefile shown in Figure 2.19 is available on the course web site under PayCalculator2.3. You can use it as a model for your own Makefiles.

**Study Questions**

2.10.1. What is a Makefile?
Chapter 3

Strings and Streams

In this chapter, we will learn about strings, I/O streams and string streams. These are not only extremely useful components of the C++ standard library, they also are great examples of abstraction and object-oriented programming.

3.1 C Strings

Strings are sequences of characters. They can be names, words, sentences or lines of text. Not surprisingly, since so many applications involve data in the form of text, strings are one of the most common types of data handled by software, second maybe only to numbers.

Our pay calculator stores the names of the input and output files as C strings. This means that each of these strings is stored in an array of characters and that the characters of the string are followed by the null character (\0).

Note that the array holding a C string can be larger than the string, which allows the string to grow and shrink by simply moving the null character. But the array still has a fixed size and this imposes a maximum length on the C string. In the next chapter, we will see that dynamic memory allocation allows us to address this problem.

The C++ library includes a class of strings that is more convenient and safer to use than C strings. We will learn about this class of strings in the next section. However, it is useful to learn to program with C strings because there are circumstances in which C strings are still used. For example, when programming in C, when working with older portions of the C++ standard library, or when implementing our own class of strings. In addition, in C++
int my_strlen( const char cs[] )
{
    int i = 0;
    while ( cs[i] != '\0' ) ++i;
    return i; // which equals the length of the string because
    // cs[i] == '\0'
}

Figure 3.1: An implementation of strlen

programs, literal strings such as "hello" are stored as C strings.

Since C strings are stored in arrays, we can work with them as we would
with any other array. But the C++ standard library provides several functions
and operators that perform useful operations on C strings. Table 3.1 lists sev-
eral of these C string operations. All are defined in the library file <cstring>,
except for the I/O functions and operators, which are defined in <iostream>. The C string functions are in the global namespace (and not in the std names-
pace like most elements of the C++ standard library.) Section A.1 describes
some additional library functions.\footnote{Some of these other functions involve concepts that we still have not covered, such as
pointers. You can ignore these functions for now.}

Note that all of these C string functions are fairly unsafe. One reason is
that they all assume that the given C strings are valid. For example, Figure 3.1
shows a possible implementation of the function strlen. (The function has
been renamed to avoid any chance of conflict with the library version.) The
function scans the array from left to right until the null character is encoun-
tered. If ever the array that holds the string does not contain a null character,
the function will go out of bounds and either return a length that makes no
sense or cause the program to crash.

Another way in which the C string functions can be unsafe has to do with
the size of the arrays. The function strcpy, for example, assumes that the
array holding dest is large enough to hold source. If the array is too small,
then strcpy will go out of bounds and overwrite other program variables
with characters from source. Or the program will crash. Either way, a bad
outcome.

The problem is that there is no reliable way for strcpy to determine the
size of the array that holds dest. The strncpy version of strcpy tries to
### 3.1. C STRINGS

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>strlen(cs)</code></td>
<td>Returns the length of C string <code>cs</code>.</td>
</tr>
<tr>
<td><code>strcpy(dest, source)</code></td>
<td>Makes C string <code>dest</code> a copy of C string <code>source</code>. The second version copying at most <code>n</code> characters. If that maximum is reached, the null character is not appended to <code>dest</code>.</td>
</tr>
<tr>
<td><code>strncpy(dest, source, n)</code></td>
<td>Same as <code>strcpy</code>, but copies at most <code>n</code> characters. If that maximum is reached, the null character is not appended to <code>dest</code>.</td>
</tr>
<tr>
<td><code>strcat(dest, source)</code></td>
<td>Appends a copy of C string <code>source</code> to C string <code>dest</code>. The second version copies at most <code>n</code> characters, followed by a null character.</td>
</tr>
<tr>
<td><code>strncat(dest, source, n)</code></td>
<td>Same as <code>strcat</code>, but copies at most <code>n</code> characters, followed by a null character.</td>
</tr>
<tr>
<td><code>strcmp(cs1, cs2)</code></td>
<td>Returns a negative integer if <code>cs1 &lt; cs2</code>, 0 if <code>cs1 == cs2</code>, a positive integer if <code>cs1 &gt; cs2</code>. Uses alphabetical order. The second version copies at most <code>n</code> characters.</td>
</tr>
<tr>
<td><code>strncmp(cs1, cs2, n)</code></td>
<td>Same as <code>strcmp</code>, but copies at most <code>n</code> characters.</td>
</tr>
<tr>
<td><code>stream &lt;&lt; cs</code></td>
<td>Outputs the characters of C string <code>cs</code>.</td>
</tr>
<tr>
<td><code>stream &gt;&gt; cs</code></td>
<td>Reads characters into C string <code>cs</code>. Skips leading white space and stops reading at white space (blank, tab or newline) or at the end of the file. The terminating character is not read.</td>
</tr>
<tr>
<td><code>stream.get(cs, n)</code></td>
<td>Reads at most <code>n-1</code> chars into C string <code>cs</code>. Does not read past the end of the current line. The null character is appended to <code>cs</code>. Does not read the newline character, if encountered.</td>
</tr>
<tr>
<td><code>stream.getline(cs, n)</code></td>
<td>Reads the rest of the current input line into C string <code>cs</code>. Does not read more than <code>n-1</code> chars. The null character is appended to <code>cs</code>. Reads the newline character, if encountered, but does not add it to <code>cs</code>.</td>
</tr>
</tbody>
</table>

Table 3.1: Some C string operations
address this problem. If \( n \) is no greater than the size of the array holding \( \text{dest} \), then \( 
strncpy \) will not go out of bounds. But if ever \( \text{source} \) is of length \( n \), then \( \text{dest} \) may be missing a null character. In addition, there is no way for \( 
strncpy \) to make sure that \( n \) is not too large. So \( 
strncpy \) is safer than \( 
strcpy \), if only because it encourages the programmer to think about the size of the array. But \( 
strncpy \) is by no means completely safe.

As mentioned earlier, literal strings such as "abc" are stored as C strings in a C++ program. Therefore, C string functions can be used on literal strings. For example, \( \text{strlen} \) can be used to compute the length of a literal string: \( \text{strlen}(\text{"abc"}) \) returns 3. And \( \text{strcpy} \) can be used to copy a literal string to another C string: \( \text{strcpy}(\text{dest, \ "abc"}) \). But note that we can’t copy anything to a literal string as in \( \text{strcpy}(\text{"abc"}, \text{source}) \). Literal strings are stored as constant C strings.

The C string functions provide good examples of procedural abstraction: to use these functions, all we need to know is what they do (and how to call them). We don’t need to know how the functions work. And there is nothing we can do that would depend on how these functions work: if ever the implementation of any these functions was changed, our code would continue to work as long as the function that was changed still does what it is supposed to do.

**Study Questions**

3.1.1. What is a C string?

3.1.2. Suppose that you want to store a string of size \( n \) as a C string. How small can the array be? How large can it be?

3.1.3. What is the difference between the C string functions \( \text{get} \) and \( \text{getline} \)?

**Exercises**

3.1.4. Create a function \( \text{println}(\text{cs}) \) that takes a C string as argument and behaves exactly as the code \( \text{cout} \ll \text{cs} \ll \text{endl} \). (But don’t use this code in implementing the function.)

3.1.5. Create a function \( \text{strlwr}(\text{cs}) \) that changes to lowercase all the letters in C string \( \text{cs} \). (You’ll probably want to use the function \( \text{tolower} \) from the standard library \( \text{cctype} \). This function takes a character as
3.2. **C++ STRINGS**

argument and returns its lowercase equivalent. If the character has no lowercase equivalent, then the function simply returns the character unchanged. You may also need to add the prefix `my` to the name of your function, as in `my_strlwr`. That’s because some cstring libraries may include the function `strlwr`.)

3.1.6. Write your own implementation of the following C string functions. Use the prefix `my` as in `my_strlen` to avoid conflicts with the library functions.

a) `strcpy`.
b) `strncpy`.
c) `strcat`.
d) `strncat`.
e) `strcmp`. Implement this function by using the comparison operators (`<`, `<=`, `==`, `>`, `>=`) on individual characters.

3.2 **C++ Strings**

The C strings of the previous section have several disadvantages. In the next chapter, we will learn how to grow these strings by dynamically allocating and deallocating arrays. But these are low-level techniques that are prone to errors. In addition, the C string library functions are not safe.

The C++ standard library includes a class of strings that addresses these problems. These strings don’t have a maximum size and they grow automatically as needed. The class is called `string`. To distinguish from C strings, objects of class `string` are often called `C++ strings`. The class `string` is defined in the library file `string` and included in the `std` namespace.

Note that C++ strings have a maximum size that is related to the largest integer that can be stored in an `int` variable. But that maximum size is typically much larger than the size of any string in most applications.²

Tables 3.2 and 3.3 show several string operations. Section A.2 describes many more.³ In these tables, the word `string` without qualifier refers to C++ strings. Note that in C++ strings, indices start at 0 just as in arrays.

---

²A typical limit is approximately 4 billion characters.
³Some of these operations involve concepts we still have not covered, such as iterators, ranges, exceptions and capacity. You can ignore these operations for now.
string s
    Creates an empty string s.

s.length()

s.size()
    Asks string s for the number of characters it currently contains.

s[i]
    Returns a reference to the character at index i in string s.

s1 = s2
    Makes string s1 a copy of string s2. The right operand can also be a C string or a single character. Returns a reference to s1.

s1 op s2
    Compares string s1 with string s2 where the operator op is one of ==, !=, <, >, <= or >=. Uses alphabetical order. Returns true or false. One of the operands must be a string object but the other can be a C string.

s1 + s2
    Returns a string that consists of a copy of string s1 followed by a copy of string s2. One of the operands must be a string object but the other can be a C string or a single character.

stream << s
    Outputs the characters of string s. Returns a reference to the stream.

stream >> s
    Reads characters into string s. Skips leading white space and stops reading at white space (blank, tab or newline). That terminating character is not read. Returns a reference to the stream.

ggetline(stream, s)
    Reads characters into string s until then end of the current line. The newline character is read but not included in s. Returns a reference to the stream.

Table 3.2: Some basic string operations
3.2. C++ STRINGS

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>string s(s2)</td>
<td>Creates a string s that is initialized to be a copy of string s2.</td>
</tr>
<tr>
<td>string s(s2, i, n)</td>
<td>A copy of the substring of s2 that starts at index i and is of length n.</td>
</tr>
<tr>
<td>string s(a, n)</td>
<td>A copy of the first n characters of array a.</td>
</tr>
<tr>
<td>string s(n, c)</td>
<td>n copies of character c.</td>
</tr>
</tbody>
</table>

The argument s2 can also be a C string.

s.empty() | Asks string s if it is empty.

s.max_size() | Asks string s for the maximum number of characters it can contain.

s.c_str() | Asks string s for a C string that contains the same characters as s.

s.substr(i, m) | Asks string s for a copy of the substring that starts at index i, and is of length m or ends at the end of the string.

s.substr(i) | 

s.resize(n) | Asks string s to change its size to n. If n is smaller than the current size of s, the last characters of s are erased. If n is larger, s is padded with the null character or with copies of character c.

s.resize(n, c) | 

s1 += s2 | Appends a copy of string s2 to string s1. The right operand can also be a C string or a single character. A reference to s is returned.

s1.swap(s2) | Asks string s1 to swap contents with string s2.

s.clear() | Asks string s to delete all its characters.

Table 3.3: Some additional string operations
The second constructor provides a way for converting C strings into C++ strings. The reverse conversion can be performed by the `c_str` method. A common use of this operation is to convert file names when those are stored as C++ strings. For example,

```cpp
ifs.open( s_file_name.c_str() );
```

In this example, the conversion is necessary because the file stream operations take C strings as arguments.

Now, you probably know that C++ functions cannot return arrays. So you may wonder how the `c_str` method can return a C string since C strings are stored in arrays. Later in these notes, we will learn that pointers can be used to return arrays in an indirect way.

As mentioned in the previous section, literal strings such as "abc" are stored as C strings in a C++ program. Therefore, any function that takes a C string as argument can be used on a literal string. For example, `string s("abc")` initializes `s` to "abc" by using the second constructor.

That second constructor can also be used for implicit conversions. Therefore, any function that takes a C++ string as argument can also be used on a C string (and a literal string). This applies to the C++ string operations themselves, as in `s + "abc"`. Note, however, that for the sake of efficiency, several operations come with separate implementations designed to handle C string arguments.

**Study Questions**

3.2.1. What are the two main advantages of C++ strings over C strings?

3.2.2. How can we convert a C string to a C++ string and vice-versa?

**Exercises**

3.2.3. Create a function `println(s)` that takes a C++ string as argument and behaves exactly as the code `cout << s << endl`. (But don’t use this code in implementing the functions.)

3.2.4. Write a code segment that starts with a string that contains a person’s name in the format "John Doe". Assume that the name contains a single blank space. Your code should produce another string that contains
3.3 I/O STREAMS

You should already be familiar with basic input and output operations in C++. The library file iostream contains two objects cin and cout that can be used for reading data from the keyboard and displaying data on the screen, as shown in Table 3.4. These objects are of class istream and ostream, respectively. These two classes provide several useful operations. Some of these are listed in Table 3.5. In that table, out and in refer to output and input streams, respectively.

File input and output is done through file stream objects of class ifstream, ofstream and fstream. The class ifstream is a subclass, or derived class, of istream, which means that all the istream operations are automatically included in ifstream. The same is true of ofstream and ostream. The class fstream is a subclass of both istream and ostream, so all the operations of these two classes are included in fstream. Table 3.6 includes some operations that are available only with file streams.

Note that if a function takes as argument an istream passed by reference, then that function can be called on any input stream, including cin as well as

<table>
<thead>
<tr>
<th>cout</th>
<th>A buffered output stream (class ostream) normally associated with the computer screen.</th>
</tr>
</thead>
<tbody>
<tr>
<td>cin</td>
<td>A buffered input stream (class istream) normally associated with the keyboard.</td>
</tr>
</tbody>
</table>

Table 3.4: Standard input and output streams

the same name but in the format "Doe, John". Do this in two different ways:

a) By using only the default constructor, the indexing operator and the methods length and resize.

b) By using only the default constructor and the methods find, push_back and append. (See Section A.2.)
<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>out &lt;&lt; data</code></td>
<td>Sends data to the output stream and returns the stream.</td>
</tr>
<tr>
<td><code>out.put(c)</code></td>
<td>Asks the output stream to write character <code>c</code>. The stream is returned.</td>
</tr>
<tr>
<td><code>in &gt;&gt; var</code></td>
<td>Reads from the input stream a value of the appropriate type and stores it in the variable. Initial white space is skipped. Returns the stream.</td>
</tr>
<tr>
<td><code>in.get()</code></td>
<td>Asks the input stream to read and return the next character.</td>
</tr>
<tr>
<td><code>in.get(var)</code></td>
<td>Asks the input stream to read the next character and store it in the variable. The stream is returned.</td>
</tr>
<tr>
<td><code>in.peek()</code></td>
<td>Asks the input stream to return the next character without removing it from the stream. The stream is returned.</td>
</tr>
<tr>
<td><code>in.putback(c)</code></td>
<td>Asks the input stream to put back character <code>c</code> so it will be the next character that is read. The stream is returned.</td>
</tr>
</tbody>
</table>

Table 3.5: Some input and output operations
3.3. I/O STREAMS

<table>
<thead>
<tr>
<th>type f</th>
</tr>
</thead>
<tbody>
<tr>
<td>Creates a file the stream of the specified type (ifstream, ofstream or fstream).</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>type f(cs_file_name)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Creates a file the stream and asks it to open the file with the given name. The argument is a C string.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>f.open(cs_file_name)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Asks file the stream to open the file with the given name. The argument is a C string.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>f.close()</th>
</tr>
</thead>
<tbody>
<tr>
<td>Asks file the stream to close the file currently associated with it.</td>
</tr>
</tbody>
</table>

Table 3.6: Some operations specific to file streams

Any input file stream. Similarly, if a function takes as argument an ostream passed by reference, then that function can be called on any output stream, including cout as well as any output file stream.

The fact that all the operations of a class like istream are also available in its subclasses is called inheritance. The fact that a reference argument declared of a certain class can be set to either an object of that class or an object of any of its subclasses is an example of polymorphism.\(^4\)

Table 3.7 shows some stream operations that are useful for performing error-checking with input and output streams. At any moment, a stream can be in any combination of the following error states: eof, fail and bad. If the stream is not in any of these error states, then it is in the good state. The stream classes provide methods to determine if a stream is in any of these states. An additional clear method is provided to return the stream to the good state. This method is useful for error recovery.

As mentioned earlier, the objects cin and cout are defined in the library file iostream. The classes istream and ostream are also defined there. The file stream classes are defined in the library file fstream. All of these classes

\(^4\)At Clarkson, the concepts of inheritance and polymorphism, as well as the mechanism for creating subclasses, are covered in detail in the course CS242 Advanced Programming Concepts.
stream.eof()
Asks the stream if it is in the eof state (an attempt was made to read past the end of the file).

stream.fail()
Asks the stream if it is in the fail state (on a previous attempt to read, the data was not in the correct format, there was no more data in the file, or the stream was not associated with a properly opened file).

stream.bad()
Asks the stream if it is in the bad state (on a previous attempt to read, an error other than the above occurred).

stream.good()
Asks the stream if it is in the good state (that is, not in any of the eof, fail or bad states).

stream.clear()
Asks the stream to return to the good state.

Table 3.7: Error-related stream operations
3.4 STRING STREAMS

and objects are part of the std namespace.

The stream objects and operations that were mentioned in this section are
sufficient for many applications. Some additional operations are described in
Section A.3.

Input and output streams are not only useful, they are also a very good
example of object-oriented programming. Consider cin. That object provides
access to the standard input stream (normally the keyboard). The cin object
most likely holds data. For example, it may hold the current contents of the
input buffer and the current location of the next character to be read. Of
course, users of cin do not depend on how that data is stored. As such, cin
can be seen as a good example of data abstraction.

But cin may not even hold that data. For example, the buffer could be
stored in some device that cin has access to. So cin is best thought of not
as a piece of data, but as an object that provides us with a service (access to
keyboard input). In addition, users of cin don’t need to be concerned with
how that service is provided. And they do not depend on those implementation
details. All of this illustrates that OOP goes beyond data abstraction and that
OOP does lead to a high level of independence.

Exercises

3.3.1. Experiment with I/O streams and the various I/O stream operations,
including error states, by writing a test driver that uses the stream ob-
jects and operations shown in Tables 3.4 to 3.7.

3.3.2. Create functions read(cs, n, in) and readln(cs, n, in) that be-
have exactly as the C string functions get and getline described in
Table 3.1. (But don’t use those functions in implementing read and
readln. Make sure your functions handle the end of a file properly.
That is, make sure they can handle reading from the last line of a file
even if that line does not end with a new line character.)

3.4 String Streams

In addition to input and output streams, C++ provides string streams.
These are streams that are each associated with a string instead of a device or
a file. For example, when you write to a string stream, characters get added
to a string. When you read from a string stream, data is read from a string.
CHAPTER 3. STRINGS AND STREAMS

Table 3.8: Some operations specific to string streams

<table>
<thead>
<tr>
<th>Operation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>type ss()</td>
<td>Creates a string stream ss of the specified type (istringstream, ostringstream or stringstream). The second version sets the string associated with the stream to be a copy of string s.</td>
</tr>
<tr>
<td>ss.str()</td>
<td>Returns a copy of the string associated with string stream ss.</td>
</tr>
<tr>
<td>ss.str(s)</td>
<td>Sets the string associated with string stream ss to be a copy of string s.</td>
</tr>
</tbody>
</table>

istringstream iss(s);
int x;
iss >> x;
++x;

ostringstream oss;
oss << x;
s = oss.str();

Figure 3.2: Adding one to the number stored in a string s

String streams support all the general stream operations described in Section 3.3. That’s because the string stream classes istream, ostream or stringstream are subclasses of the I/O stream classes istream, ostream and iostream. Table 3.8 lists some operations that are particular to string streams. (These operations are also listed in Section A.4.)

An important application of string streams is performing conversions between strings and other data types. For example, Figure 3.2 shows a code fragment that takes a string s containing the digits a number and adds 1 to that number, in the sense that if s contained the digits of 34, it now contains the digits of 35.

This works by first creating an input string stream and setting its associated
string to be a copy of $s$. An integer is then read from the string stream. This essentially converts the string into an integer. The integer is then incremented and written back to an output string stream. This performs the conversion of the integer back to a string. The string associated with the output string stream is then copied to $s$.

The string stream classes are defined in the library file `sstream`. These classes are part of the `std` namespace. Source code for the example of Figure 3.2 is available on the course web site as `stringstreams.cpp`.

**Exercises**

3.4.1. Write a code fragment that reads a line of text from `cin` and verifies that the line contains exactly three integers and nothing else. *Hint*: The code `cin >> x >> y >> z` does not work because the three integers may come from different input lines.
Chapter 4

Error Checking

In this chapter, we will discuss the detection and handling of errors. In particular, we will learn how to use exceptions and how to work with stream error states.

4.1 Introduction

The pay calculator we created in the first two chapters of these notes is reliable: it does what it is supposed to do when it is used properly. Ideally, we would also like the program to be robust or fail-safe: we would like it to always behave in a reasonable way, no matter what. Essentially, this means that our program should test for possible errors and contain code to deal with these errors.

The errors that a program should guard against can be characterized as either internal or external. Internal errors are errors in the program itself. External errors are usually errors in the input given to the program. This could be badly formatted data, data with invalid values, or files that don’t open.

To detect errors in the program itself, it helps to have functions check that their preconditions are satisfied, whenever this is possible and practical. The preconditions of a function are the conditions that must exist at the time a function is called for that function to be able to do its job properly.

For example, the set_hours method of class Time requires that its argument new_hours be a valid number of hours, that is, an integer from 0 to 23. We could consider having the method test to verify that this is indeed the
case:

        if ( new_hours >= 0 && new_hours < 24 )

However, this test would take longer than the main job of the method, which is to set the data member hours:

        hours_ = new_hours;

So the additional safety of having set_hours check its argument comes at a significant price in terms of running time.

A reasonable approach is to have functions perform error-checking when it is either cheap or critical. Checking arguments is often cheap when a function has a complex task to perform. Error-checking is usually critical when a function reads data from the user or an input file.

Besides deciding whether a function should test for errors, we also need to decide what a function should do when it detects an error. As a general rule, it is usually best to have the function try to recover or, if that can’t be done, to simply let the calling function know that there was a problem. That function is usually in a much better position to decide on the appropriate course of action.

There are many ways in which a function can let its calling function know that an error occurred. For example, the set_hours method could set the time to an invalid value, return the value false, or set an additional argument to false. Those three options are illustrated in Figure 4.1.

Note that if the first option is chosen, then it is useful to include in the class a method that tests whether the time is valid, as shown in Figure 4.2.

The first two options (making the time invalid and returning a Boolean flag) have a significant weakness: it is very easy for the calling function to ignore the error, either by accident or by laziness. The third option makes it harder to ignore the error because it requires the explicit creation and passing of a Boolean variable.

But it is still possible for the calling function to omit to test that Boolean argument after the function returns. In addition, with some functions, it is not possible to add extra arguments. This is often the case with operators. Default constructors are another example. The next section will introduce a mechanism for error reporting that addresses these issues.
void set_hours( int new_hours ) {
    if ( new_hours >= 0 && new_hours < 24 )
        hours_ = new_hours;
    else
        hours_ = 99;
}

bool set_hours( int new_hours ) {
    if ( new_hours >= 0 && new_hours < 24 ) {
        hours_ = new_hours;
        return true;
    }
    else {
        return false;
    }
}

void set_hours( int new_hours, bool & success ) {
    if ( new_hours >= 0 && new_hours < 24 ) {
        hours_ = new_hours;
        success = true;
    }
    else {
        success = false;
    }
}

bool is_valid() const {
    if ( hours() >= 0 && hours() < 24 )
        return true;
    else
        return false;
}

Figure 4.1: Different ways of reporting errors

Figure 4.2: The method is_valid
Study Questions

4.1.1. What is a reliable program? What is a fail-safe or robust program?

4.1.2. What is the main weakness of error flags as a mechanism for reporting errors?

Exercises

4.1.3. Add a method `is_valid` to each of the classes `Date` and `Fraction` you created for the exercises of Chapter 2.

4.2 Exceptions

Exceptions provide a way of reporting errors that is more robust than return values or arguments. The idea is that when a function needs to report an error to its calling function, it *throws* an exception. When an exception is thrown, control is immediately returned to the calling function. That function then has two options:

1. **Catch** the exception and deal with the error.
2. Ignore the exception, which causes the calling function to throw the same exception.

An exception that is not caught will continue to propagate all the way to `main`. If `main` does not catch the exception, then the program aborts. It is therefore impossible for an exception to be completely ignored.

An exception can be a value of any type. However, exceptions are usually objects that carry information about the error that occurred. Even when an exception object doesn’t contain data, the class of the exception is already useful information since it can be used to determine the appropriate way of dealing with the error.

We will illustrate the use of exceptions with the `set_hours` method of class `Time`. Figure 4.3 shows an exception class `TimeError`. Each of these exception objects holds a description of the error. The class includes a constructor that allows the description to be initialized when the exception object is created. It also includes a method `what` that allows the description to be retrieved.
class TimeError
{
public:
    TimeError( const std::string & d ): description(d) {}
    const std::string & what() const { return description; }
private:
    std::string description;
};

Figure 4.3: The exception class TimeError

void set_hours( int new_hours ) throw(TimeError) {
    if ( new_hours >= 0 && new_hours < 24 ) {
        hours_ = new_hours;
    } else {
        throw TimeError(
            "Invalid argument given to Time::set_hours";
        )
    }
}

Figure 4.4: A set_hours method that throws exceptions

Figure 4.4 shows a revised set_hours method that verifies that its argument is a valid number of hours before using it to set the Time object. In case the argument is invalid, a TimeError is thrown.

Note how the header of the method contains a throw declaration. The purpose of that declaration is not to allow the method to throw the exception; functions without a throw declaration are allowed to throw any exception. On the contrary, the throw declaration restricts the function to be able to throw only the listed exceptions. This is a safety measure to prevent the accidental throwing of the wrong type of exception.

Figure 4.5 shows a test driver for set_hours. For exceptions to be caught, the method must be used within a try statement and that statement must be followed by a catch block that contains code for handling the error.

Note that we have included in the try statement not only a call to the set_hours method but also all the code that would normally execute if no error occurred during the execution of set_hours. While this is not absolutely
try {
    cout << "Setting hours to " << h << "...\n";
    t.set_hours(h);
    cout << "Success!\n";
}
catch (const TimeError & e) {
    cout << "Error: " << e.what() << endl;
}
cout << t << endl;

Figure 4.5: Testing set_hours

necessary, it tends to produce simpler and more readable code. If an exception is thrown at any point inside the try statement, execution of the try statement is immediately halted and execution resumes with the code that follows the try statement.

The main advantage of exceptions, as we mentioned earlier, is that they are essentially impossible to ignore. Another advantage is that they neatly separate the main code (inside the try statement) from the error handling code (inside the catch block).

But exceptions also have disadvantages. First, the separation of the main code from the error handling code means more jumping around in the normal flow of execution, possibly making the code harder to understand. Second, if a function is modified to make it throw exceptions, then the calling function needs to be modified too, to catch the exception. But if we forget, the compiler will not complain. This would not be the case if an additional argument had been added to the function being called. Finally, serious errors can be introduced in a program when constructors throw exceptions, a point that we will cover in more detail later in these notes.

There is some debate whether the advantages of exceptions outweigh their disadvantages. Some say that exceptions are great but too hard to program correctly. Either way, whether exceptions are used in a project is not up to each individual programmer. It is therefore important to be familiar with them.

Source code for the revised class Time and a test driver is available on the course web site under Time2.0.
4.3. INPUT VALIDATION

Study Questions

4.2.1. What is an exception?

4.2.2. What can a function do when it calls another function and that other function throws an exception?

4.2.3. What happens if main fails to catch an exception?

4.2.4. What are two advantages and three disadvantages of exceptions?

Exercises

4.2.5. Revise the method set_minutes of class Time so it throws a TimeError in case its argument is invalid.

4.2.6. Revise the set methods of the classes Date and Fraction you created for the exercises of Chapter 2 so they throw exceptions in case their arguments are invalid. Create exception classes similar to TimeError for each of Date and Fraction.

4.3 Input Validation

Input validation, which is to verify that input data is free from errors, is critical for quality software. But it is also one of the most tedious programming tasks, largely because there are usually so many ways in which input data can be invalid. To illustrate this, consider our class Time. The valid input format is h:mm or hh:mm where each h and m represents a single digit. Let’s simplify the error checking by not insisting that the number of hours and minutes consist of the specified number of digits. For example, we’ll accept 008:5 and interpret it to be 8:05.

In what follows, we will need to add error states to a stream. (Recall that a stream can be in more than one error state at a time.) This can be done by using the method setstate(errorbit) where the argument is one of the three values eofbit, failbit or badbit. These values are contained within the stream object itself and therefore can be accessed as stream.eofbit, stream.failbit and stream.badbit.

Figures 4.6 and 4.7 show how we can read a time t from input stream in while checking the validity of the input data. We start by reading an integer
using the standard input operator: \texttt{in >> h}. What could go wrong? After the reading of \texttt{h}, the stream could be in an error state. If the state is fail or bad, then the stream didn’t contain an integer so we quit and return the stream as is. If the state is eof but not fail, then an integer was read but it was at the very end of the file associated with \texttt{in}. This implies that the stream is missing the colon and the minutes so it cannot possibly contain a valid time. In that case, we also quit but we add the fail state to the stream. If the stream is not in an error state, then we check the value of \texttt{h}. If it’s invalid, we quit and return the stream after setting its state to fail.

All this work was just for reading the hours. We must then read a character, make sure it’s a colon, check that there is no white space after the colon, read the minutes and check that they’re valid. And we must do all that while testing for every possible thing that could go wrong.

And recall that we are not even checking for all possible errors. In addition, the current version of the input operator is \textit{destructive}, in the sense that in case of an error, some data will be read from the stream and lost. A nondestructive input operator would need to read characters one at a time so that they can be restored in case of an error.

Note that we chose to have the input operator report errors through the error state of the stream. This is the standard approach. It has the advantage that other developers are already familiar with it. One weakness, however, is that little information is returned about the exact cause of the error. Exceptions could be used to address this.

We can now add error-checking to the pay calculator program. It will use the error-checking performed by the input operator of class \texttt{Time} to check for errors in the input file. Figures 4.8 and 4.9 show a revised \texttt{run_pay_calculator} function that now delegates some of its work to a new \texttt{process_one_line} function. Lines are read one at a time and then an employee number, a start time and a stop time are extracted from each line. If any of these extractions fail, an error message is printed. If the line contains superfluous data, a warning is printed.

Source code for a class \texttt{Time} with the input operator presented in this section is available on the course web site under \texttt{Time2.1}. The revised pay calculator is available under \texttt{PayCalculator2.4}. 
4.3. INPUT VALIDATION

```cpp
istream & operator>>( istream & in, Time & t ) {
    // read hours after skipping whitespace
    int h = 99;
    in >> h;
    // Note: we don’t read directly into t.hours. so we can leave
    // it unchanged in case of an error.
    if ( in.fail() || in.bad() ) return in; // no int in stream
    if ( in.eof() || h < 0 || h > 23 ) {
        // if eof, an int but nothing else
        in.setstate(in.failbit);
        return in;
    }
    // make sure next char is colon
    char next_char = in.peek();
    // Note: we don’t read the char until we’re sure it’s a
    // colon.
    if (!in ) return in;
    // Note: bad or good were the only possibilities.
    if ( next_char != ':' ) {
        in.setstate(in.failbit);
        return in;
    }
    in.get(); // read colon
}
```

Figure 4.6: Input operator that performs error checking (part 1 of 2)
CHAPTER 4. ERROR CHECKING

```cpp
istream & operator>>( istream & in, Time & t ) {
    ...

    // make sure next char is not whitespace
    next_char = in.peek();
    if ( !in ) return in;
    if ( next_char == ' ' || next_char == '	' || next_char == '
' ) {
        in.setstate(in.failbit);
        return in;
    }

    // read minutes
    int m = 99;
    in >> m;
    if ( in.fail() || in.bad() ) return in;
    // Note: eof is possible if the minutes were at the very
    // end of the file. This shouldn’t make the time invalid
    // but we leave the stream in the eof state.
    if ( m < 0 || m > 59 ) {
        in.setstate(in.failbit);
        return in;
    }

    // all good (except possibly for the number of digits in
    // the hours and minutes)
    t.set_hours(h);
    t.set_minutes(m);
    return in;
}
```

Figure 4.7: Input operator that performs error checking (part 2 of 2)
4.3. INPUT VALIDATION

void run_pay_calculator() {
    ifstream ifs_times( kcsTimesFileName );
    ofstream ofs_report( kcsReportFileName );

    cout << "Reading 'hours.txt'...\n\n";

    string s_line;
    int line_number = 0;
    int num_errors = 0;
    int num_warnings = 0;

    while ( getline(ifs_times, s_line) ) {
        ++line_number;
        process_one_line(s_line, line_number,
                         ofs_report, num_errors, num_warnings);
    }
    ifs_times.close();
    ofs_report.close();

    if ( num_errors != 0 || num_warnings != 0 ) cout << endl;
    cout << num_errors << " error(s) and "
        << num_warnings << " warning(s).\n\n" << "Output written to 'reports.txt'.\n\n";
}

Figure 4.8: Revised run_pay_calculator function (part 1 of 2)
void process_one_line(const string & s_line, int line_number, ofstream & ofs_report, int & num_errors, int & num_warnings) {
    // read start and stop times
    istringstream iss_line(s_line);
    int employee_number = -1;
    Time start_time, stop_time;
    string s_error_message;
    if ( !(iss_line >> employee_number) )
        s_error_message = "unable to read employee number";
    else if ( !(iss_line >> start_time) )
        s_error_message = "unable to read start time";
    else if ( !(iss_line >> stop_time) )
        s_error_message = "unable to read stop time";

    if ( s_error_message.empty() ) { // no reading error
        // check that rest of line is blank
        iss_line >> ws;
        if ( !iss_line.eof() ) {
            ++num_warnings;
            cout << "Warning: "
                << "detected improper data at end of line "
                << line_number << endl;
        }
    }

    // compute and print pay
    double pay = ( stop_time - start_time ) * kPayRate;
    ofs_report << employee_number << ' ' << start_time << ' ' << stop_time << "$" << fixed
        << setprecision(2) << pay << endl;
} else { // there was a reading error
    ++num_errors;
    cout << "Error: " << s_error_message << " at line "
        << line_number << endl;
}

Figure 4.9: Revised run_pay_calculator function (part 2 of 2)
4.3. INPUT VALIDATION

Study Questions

4.3.1. What method can be used to add an error state to a stream? What values does this method take?

Exercises

4.3.2. Add error-checking to the input operator of the class Date you created for the exercises of Chapter 2. The operator should check that the date is entered in the format m/d/y where m is a number from 1 to 12, d is a number from 1 to 30, and y is an integer. Spaces are not allowed on either side of the slashes (/). Use error states as was done for Time in this section.

4.3.3. Add error-checking to the input operator of the class ThreeDVector you created for the exercises of Chapter 2. The operator should check that the vector is entered in the format (x, y, z) where x, y and z are real numbers. Any number of spaces is allowed between the parentheses, commas and numbers. Use error states as was done for Time in this section.

4.3.4. Add error-checking to the input operator of the class Fraction you created for the exercises of Chapter 2. The operator should check that the fraction is entered in the format a/b where a is an integer and b is a positive integer. Spaces are not allowed on either side of the slash (/). Use error states as was done for Time in this section.

4.3.5. Modify the Time input operator to throw a TimeError in addition to setting an error state for the stream. Include in the exception object a description of the error. Be as precise as possible.

4.3.6. Make the Time input operator nondestructive and fully robust. In particular, it should check that the hours and minutes have the specified number of digits.
Chapter 5

Vectors

In this chapter, we turn our focus to the storage of data. We will learn to use vectors, a type of container provided in the C++ standard library. Vectors are essentially improved arrays. As a motivating application, we will create a simple file viewer program.

5.1 A Simple File Viewer

We will create a simple file viewer that allows the user to view the contents of a text file. The file viewer is not an editor: the user can only view the contents of the file, not modify it.

Figure 5.1 shows what the interface of this program might look like. The program displays a certain number of lines from the file and a menu of commands. The commands next and previous cause the file viewer to display the next or previous “pages”. The command open causes the program to ask the user for the name of another file.

We will design and implement this program later in this chapter. The main issue in the design of the file viewer is the storage of the file contents. To avoid having to read the file multiple times, we will have the program store the contents of the file in main memory, that is, in the program’s variables.

How should we store the contents of the file? One option is to use an array of strings, with each string holding the contents of one line. But arrays have a significant weakness: their size must be a number that is determined at compile-time, before the program runs. This would limit the file viewer to files that are no larger than the size of the array. To handle larger files, the
After a few computer science courses, students may start to get the feeling that programs can always be written to solve any computational problem. Writing the program may be hard work. For example, it may involve learning a difficult technique. And many hours of debugging. But with enough time and effort, the program can be written.

So it may come as a surprise that this is not the case: there are computational problems for which no program exists. And these are not ill-defined problems (Can a size of the array would need to be increased and the program recompiled. In this chapter, we will learn that vectors provide a much more flexible approach.

Study Questions

5.1.1. What is a significant weakness of (ordinary) C++ arrays?

5.2 Arrays and Vectors

As mentioned in the previous section, ordinary C++ arrays have a predetermined size. (In this context, predetermined means determined at compile time, before the execution of the program.) The size of an array is also fixed: it cannot change during the execution of the program. This is a significant limitation that can make a program fail in case an array is too small, or waste memory in case an array is unnecessarily large.

A solution to this problem is to work with dynamically allocated arrays. We will learn how to do this in the next chapter. However, C++ arrays have
void print( const int a[], int n )
// Prints the contents of an array of ints. n is the size of
// the array.
{
    for ( int i = 0; i < n; ++i ) cout << a[i] << ' ';
    cout << endl;
}

Figure 5.2: A function that prints an array

several other weaknesses, even when they are dynamically allocated.

First, arrays are not aware of their size. In particular, there is usually no
reliable way for a function to figure out the size of an array argument, which is
why we typically pass the size of the array as a separate additional argument.
For example, Figure 10.3 shows a function that prints the contents of an array
of integers. One danger is that nothing guarantees that the value of that size
argument is correct.

Second, the usual operators, such as =, == and <, don’t work with arrays.
Actually, they can sometimes work but they don’t do what you would expect.
For example, if a and b are two arrays, then a = b, if it compiles, will cause
the names a and b to refer to the same array. This is called a shallow copy. In
contrast, we would likely want a = b to cause a to become a separate copy of
array b. This is called a deep copy. With arrays, dynamically allocated or not,
the only way to achieve a deep copy is to write a loop. This is inconvenient
and a possible source of errors.

Third, it is not easy to have a function return a copy of an array. In
particular, the return type of a C++ function cannot be an array. There are
ways to get around this problem, by using dynamically allocated arrays and
other techniques we will learn in the next chapter. But these techniques are
inconvenient and prone to errors.

To summarize, C++ arrays have the following four weaknesses:

1. Arrays have a fixed, predetermined size.
2. Arrays don’t know their size.
3. Arrays don’t support the usual operators.
4. It is not easy to have a function return a copy of an array.
As mentioned earlier, the first weakness can be addressed by dynamically allocating the arrays. However, dynamically allocated arrays involve low-level techniques that can be difficult to program correctly. And they still have the other three weaknesses.

Vectors have been developed precisely to address all of these weaknesses. Just like an array, a vector is an object that stores a sequence of elements all of the same type.

The first weakness is addressed by having each vector store its elements in a dynamically allocated array. The size of that array can be determined at run time and the array can be replaced by a larger one if needed, meaning that the size of the vector is neither fixed nor predetermined.

That dynamically allocated is a private member of the vector, which means that the array and the low-level techniques related to dynamic memory allocation are hidden from the user of the vector. This is a good example of the benefits of data abstraction.

The second weakness is addressed by having each vector store its size in a data member. And to ensure that the size data member always holds the correct value, it is private. This illustrates how encapsulation (the hiding in one construct of several pieces of data) can be used to ensure the integrity of data. We already saw that encapsulation can be used to enforce data abstraction. Data integrity is another benefit.

Vectors provide several methods, including the following three:

- An indexing operator so that the vector elements can be accessed just like the elements of an array.
- A method that returns the size of the vector so the user can retrieve that size whenever needed.
- A method that resizes the vector so that vectors have the flexibility of dynamically allocated arrays.

To address the third weakness, vectors overload the assignment operator (=) as well as the comparison operators (such as == and <).

The fourth weakness is addressed somewhat automatically because when a function returns a vector, the vector is automatically copied, just like any other object or value (except arrays).

The C++ standard library includes a class of vectors. We will learn how to use this class in the rest of this chapter. In the next two chapters, we will learn
the techniques needed to implement such a class. This will allow us to learn techniques that are useful in the implementation of many data structures. In addition, this will give us a deeper understanding of vectors, which should help us use them more effectively.

Study Questions

5.2.1. What are three different advantages of vectors over dynamically allocated arrays?

5.2.2. What is an additional advantage of vectors over ordinary arrays?

5.3 Vectors in the STL

As was just mentioned, the C++ standard library includes a class of vectors. This class belongs to a portion of the library called the *Standard Template Library* (STL). STL classes that implement data structures are called *containers*.

The STL class of vectors is simply called **vector**. Just like arrays, STL vectors are **generic** in the sense that they can hold elements of any type (but with the restriction that each vector holds elements of a single type).

Tables 5.1 and 5.2 show some of the most basic vector operations. Section A.5 describes several other operations.\(^1\)

Note that the **swap** method is not only convenient, it is also very efficient: the contents of the two vectors are swapped without any elements being copied. Note also that STL vectors have **push_back** and **pop_back** methods but no **push_front** or **pop_front**. The reason has to do with efficiency: it is possible to implement the operations at the back efficiently but doing so at the front is more difficult. (We will discuss this point in more detail later in these notes.)

The **vector** container is defined in library file **vector** and included in the **std** namespace.

Study Questions

5.3.1. In what way are vectors generic?

\(^1\)Some of these operations involve concepts we still have not covered, such as iterators, ranges and capacity. You can ignore these operations for now.
vector<T> v
vector<T> v(n)
vector<T> v(n, e)
vector<T> v(v2)

Creates a vector \( v \) that can hold elements of type \( T \). The vector is initialized to be empty, or to contain \( n \) copies of the default object of class \( T \), or \( n \) copies of element \( e \), or to be a copy of vector \( v2 \).

v.size()

Asks vector \( v \) for the number of elements it currently contains.

v[i]

Returns a reference to the element at index \( i \) in vector \( v \).

v.front()
v.back()

Asks vector \( v \) for a reference to its front or back element.

v.resize(n)
v.resize(n, e)

Asks vector \( v \) change its size to \( n \). If \( n \) is smaller than the current size of \( v \), the last elements of \( v \) are deleted. If \( n \) is larger than the current size, then \( v \) is padded with either copies of the default object of class \( T \) or with copies of element \( e \).

v.push_back(e)

Asks vector \( v \) to add a copy of element \( e \) to its back end.

v.pop_back()

Asks vector \( v \) to delete its last element.

v.clear()

Asks vector \( v \) to delete all its elements.

v1 = v2

Makes vector \( v1 \) a copy of vector \( v2 \). Returns a reference to \( v1 \).

Table 5.1: Some basic vector operations
5.4. IMPLEMENTATION OF THE FILE VIEWER

| v.empty() | Asks vector v if it is empty. |
| v.max_size() | Asks vector v for the maximum number of elements it can contain. |
| v.assign(n, e) | Asks vector v to change its contents to n copies of element e. |
| v1.swap(v2) | Asks vector v1 to swap contents with vector v2. |

Table 5.2: Some additional vector operations

5.3.2. Why do STL vectors have no push_front or pop_front methods?

Exercises

5.3.3. Experiment with vectors by writing a test driver that creates more than one type of vector and uses all the methods shown in Tables 5.1 and 5.2.

5.4 Implementation of the File Viewer

We can now use STL vectors to create the file viewer program we outlined at the beginning of this chapter. As usual, it is a good idea to create this program gradually. Figure 5.3 shows the main function of a first version of the program. In this version, the entire contents of the file is displayed (not just a certain number of lines). Figure 5.4 shows the auxiliary functions of the program.

Figure 5.5 shows the main function of a second version of the program. (Portions that are identical or nearly identical to the first version have been replaced by dots to allow the code to fit on one page.) In this version, only ten lines are displayed. Note how the implementation of the next and previous commands guarantees that these commands do not move too far in either direction.

Figure 5.6 shows the auxiliary functions of the second version of the program. The implementation of display takes care not to access lines that are
void run_file_viewer() {
    vector<string> v_document_lines;
    string file_name;

    open_file(file_name, v_document_lines);

    while (true) { // while command is not 'quit'
        display(file_name, v_document_lines);

        cout << "command: ";
        char command = '-';
        cin.get(command);
        cin.get(); // '\n'

        switch (command) {
            case 'q': return;
            case 'o': {
                open_file(file_name, v_document_lines);
            }
        }
    }
}

Figure 5.3: The main function of a first version of the file viewer
5.4. IMPLEMENTATION OF THE FILE VIEWER

```cpp
void display(const string & file_name, const vector<string> & v_document_lines) {
    cout << endl << file_name << endl;
    string long_separator(50, '-');
    cout << long_separator << endl;
    for (int i = 0; i < v_document_lines.size(); ++i)
        cout << setw(3) << i+1 << " " << v_document_lines[i] << endl;
    cout << long_separator << endl
        << " open quit" << endl;
    string short_separator(8, '-');
    cout << short_separator << endl;
}

void open_file(string & file_name, vector<string> & v_document_lines) {
    cout << "file: ";
    getline(cin, file_name);
    ifstream ifs_document(file_name.c_str());
    v_document_lines.clear();
    string line;
    while (getline(ifs_document, line))
        v_document_lines.push_back(line);
    ifs_document.close();
}
```

Figure 5.4: The auxiliary functions of the first version of the file viewer
const int windowHeight = 10;

void run_file_viewer() {
    string file_name;
    vector<string> v_document_lines;
    int ix_current_line = 0;

    open_file(file_name, v_document_lines, ix_current_line);

    while (true) { // while command is not 'quit'
        display(file_name, v_document_lines, ix_current_line);

        ...

        switch (command) {
            ...

            case 'n': {
                if (ix_current_line + windowHeight < v_document_lines.size())
                    ix_current_line += windowHeight;
                break;
            }

            case 'p': {
                if (ix_current_line - windowHeight >= 0)
                    ix_current_line -= windowHeight;
                break;
            }
        }
    }
}

Figure 5.5: The main function of a second version of the file viewer
void display(const string & file_name,
        const vector<string> & v_document_lines,
        int ix_current_line) {
    cout << endl << file_name << endl;
    string long_separator(50, '-');
    cout << long_separator << endl;
    int ix_start_line = ix_current_line;
    int ix_stop_line = ix_current_line + windowHeight;
    for (int i = ix_start_line; i < ix_stop_line; ++i) {
        cout << setw(3) << i+1 << " ";
        if (i < v_document_lines.size())
            cout << v_document_lines[i];
        cout << endl;
    }
    cout << long_separator << endl
        << " next previous open quit" << endl;
    string short_separator(8, '-');
    cout << short_separator << endl;
}

void open_file(string & file_name,
        vector<string> & v_document_lines,
        int & ix_current_line) {
    cout << "file: ";
    getline(cin, file_name);
    ifstream ifs_document(file_name.c_str());
    v_document_lines.clear();
    ix_current_line = 0;
    string line;
    while (getline(ifs_document, line))
        v_document_lines.push_back(line);
    ifs_document.close();
}

Figure 5.6: The auxiliary functions of the second version of the file viewer
not present in the document.

Source code for both versions of the file viewer is available on the course
web site under FileViewer.

Exercises

5.4.1. Modify the file viewer as described below. Modify the program as little
as possible.

a) Instead of having the window height always be 10, the program
 begins by asking the user for the number of lines to be displayed
 on each “page”.

b) Add a jump command that asks the user for a line number and
 redisplays the document with the requested line at the top. Hint:
 Make sure that the previous command is still able to reach the
 beginning of the document after a jump.

c) Add a search command that asks the user for a piece of text and
 redisplays the document so that the first line that contains the text
 is displayed at the top of the window. The search begins at the
 current line and proceeds to the end of the document. Hint: Use
 the string search methods described in Section A.2.
Chapter 6

Dynamically Allocated Arrays

As discussed in the previous chapter, ordinary C++ arrays have a number of weaknesses, including the fact that their size must be fixed and determined at compile time. In this chapter, we will learn to address this problem through dynamic memory allocation. In the process, we will also introduce the concept of a pointer. These concepts will play a key role in the implementation of vectors in the next chapter.

6.1 The Size of Ordinary Arrays

Before learning how to create arrays whose size is determined at run time, it is useful to first understand why the size of an ordinary C++ array must be a constant that’s determined at compile time. This requires taking a brief look at the management of data during the execution of a program.

During the execution of a program, when a function is called, memory space is automatically allocated for all the local variables of the function. The values of these variables are normally stored in a block of memory called an *activation record*. In an activation record, the values of the variables are usually stored side-by-side. Without getting into all the details, the advantage of this setup is that it allows the values of the variables to be accessed very efficiently at run time.

But to gain full advantage of this setup, the compiler must know the size of each variable. This means that variable sizes must be determined before the

\[1\text{At Clarkson, courses such as CS241 *Computer Organization* and CS445 *Compiler Construction* normally cover this subject in more detail.}\]
program is compiled. In addition, these sizes cannot change since each variable
is stored in a fixed amount of space somewhere in an activation record. These
restrictions apply to all the local variables of a function, including arrays.

These restrictions also applies to the data members of a class or structure. For
example, when an object is created, a block of memory is allocated to store
the values of the data members of the object. In this block of memory, the
data members are stored side-by-side, just like the local variables of a function
are stored side-by-side in an activation record. Once again, this setup allows
the values of the data members to be accessed very efficiently but it requires
that each data member have a fixed size that’s determined at compile time.

Therefore, if an array is going to have a size that’s determined at run time,
the array will have to be stored somewhere else, not in the activation record
of a function (and not inside an object). We will learn how to do that in the
next section. The price we will have to pay is that accessing the elements of
these arrays will take slightly more time.

Study Questions

6.1.1. What is the advantage of storing variables side-by-side in an activation
record? What is the disadvantage?

6.2 The Dynamic Allocation of Arrays

In previous section, we learned that for an array to have a size that’s deter-
mined at run time, the array must be stored outside the activation record of
a function, and outside of any object. This can be done by using the new
operator as in

    new int[n]

In this example, the new operator allocates, or reserves, a block of memory
large enough to store an array of n integers.

That block of memory doesn’t have a name, like an ordinary array. So how
do we access it? Each location in a computer’s memory has an address that
uniquely identifies it and can be used to access it. The new operator returns
the address of the allocated block of memory. This address can be stored in a
special variable called a pointer as in

    int * da = new int[n];
6.2. THE DYNAMIC ALLOCATION OF ARRAYS

```cpp
int n = 0;
cin >> n;

int * da = new int[n];

for (int i = 0; i < n; ++i) da[i] = i*10;

for (int i = 0; i < n; ++i) cout << da[i] << ' ';
```

Figure 6.1: Creating and accessing a dynamically allocated array

The * indicates that da is a pointer to an integer, not an actual integer. And the type of the pointer must match the type of the array. For example,

```cpp
string * da = new string[n];
```

creates an array of n strings and set da to point to it.

Note that da is declared as a pointer to a single value (of type int or string, in the above examples). When allocating an array, new returns a pointer to the first element of the array. That pointer can then be used to access all the elements of the array by using the usual indexing operator as in da[i]. For example, Figure 6.1 shows the creation, initialization and printing of an array. Note how the size of the array is determined at run time by the user of the code.

The local variables of a function are automatically allocated when the function is called. Similarly, the data members of an object are automatically allocated when the object is created. This is called automatic memory allocation and these variables are called automatic variables.

In contrast, variables created by the new operator are said to be dynamically allocated and they are called dynamic variables. As we mentioned before, dynamic variables do not have names, they only have addresses.

In this section, we used da as a generic name for a dynamically allocated array. We will also use da as a variable name prefix to indicate that the variable points to a dynamically allocated array. In contrast, for ordinary arrays, we will use the prefix a.

Study Questions

6.2.1. What does the new operator do?
// n is the current size of da
int * da_new = new int[2*n];
for ( int i = 0; i < n; ++i ) da_new[i] = da[i];
delete [] da;
da = da_new;
da_new = NULL;

Figure 6.2: Resizing a dynamically allocated array

6.2.2. When allocating an array, what value does the new operator return?

6.2.3. What is an automatic variable?

6.2.4. What is a dynamic variable?

6.2.5. What is the main advantage of dynamically allocated arrays?

6.3 Programming with Dynamically Allocated Arrays

In this section, we look at several important issues that come up when programming with dynamically allocated arrays. These include the resizing, copying and deallocation of dynamically allocated arrays, as well as the NULL pointer value and the passing of dynamically allocated arrays as arguments to functions.

We now know how to dynamically allocate arrays so their size can be determined at run time. But can that size change? Strictly speaking, no. However, a pointer to a dynamically allocated array can be made to point to another dynamically allocated array. This allows us to indirectly resize a dynamically allocated array as illustrated in Figure 6.2.

This code starts by allocating an array da_new that’s double the size of the array da. (Note how we are using the name of the pointers as the names of the arrays too, even though, strictly speaking, the arrays have no name.) Then, the elements of da are copied to da_new.

Note that da_new = da would not work. This would only copy the pointer da, not the array that da points to. That is, it would make da_new point the
array that \(da\) points to. It would not copy the elements of the array that \(da\) points to into the larger array that \(da_{\text{new}}\) points to.

Once the elements are copied, the old array is deallocated so that the memory space that it uses can be later used to store other variables. This is done with the \texttt{delete} operator. The square brackets specify that an entire array is to be deallocated, not just the one integer that \(da\) points to. If we neglected to deallocate the old array, the memory space would continue to be reserved for the array even though we would no longer have any way of accessing that memory space. This would reduce the amount of memory available to the program and would be called a \textit{memory leak}.

After the old array is deallocated, \(da\) is set to point to the new array. Finally, \(da_{\text{new}}\) is set to the special pointer value \texttt{NULL}, which is essentially a way of making \(da_{\text{new}}\) point to nothing.\footnote{In fact, \texttt{NULL} is the address of some memory location but that memory location is usually out of bounds. This means that if a \texttt{NULL} pointer is used to access that memory location, the program is essentially guaranteed to crash.} This is done as a precaution so that \(da_{\text{New}}\) is not used later to accidentally modify or deallocate the array. The \texttt{NULL} value is defined in the standard libraries \texttt{cstddef} and \texttt{cstdlib}.

The passing of dynamically allocated arrays as arguments to functions raises a few particular issues. Figure 6.3 shows two functions \texttt{init} and \texttt{print} that initialize and print the contents of an array. The array is passed to the functions by simply passing a pointer to its first element.

Note that the array argument could have also been declared as \texttt{int \(a[]\)}. This is equivalent to \texttt{int \* \(a\)}. Note also that the argument \(a\) of \texttt{print} is declared as pointing to a constant integer. This prevents the function from modifying the contents of the array.

The functions \texttt{init} and \texttt{print} can be used on ordinary arrays as well as on dynamically allocated arrays, as in

\begin{verbatim}
int a[5];
init(a, 5);
print(a, 5);
\end{verbatim}

(This is the reason why we called the argument of these functions \(a\) instead of \(da\).) This works because an ordinary array is automatically converted to a pointer to its first element whenever needed.

Here is another example of this kind of conversion:

\begin{verbatim}
int * b = a;
\end{verbatim}
void init( int * a, int n )
// Initializes an array of ints to contain 0, 1, 2, ..., (n-1).
// a points to the array and n is the size of the array.
{
    for ( int i = 0; i < n; ++i ) a[i] = i;
}

void print( const int * a, int n )
// Prints the contents of an array of ints. a points to the
// array and n is the size of the array.
{
    for ( int i = 0; i < n; ++i ) cout << a[i] << ' ';
    cout << endl;
}

Figure 6.3: Functions that initialize and print an array

This makes the pointer \( b \) point to array \( a \) (to the first element of array \( a \)). Then \( b \) can be used to access the array, just as if it was the name of the array. For example,

```
for ( int i = 0; i < 5; ++i ) b[i] = 0;
print(a, 5);
```

fills the array with 0's and then prints its contents.

Figure 6.4 shows a function that resizes a dynamically allocated array. The argument \( da \) is a pointer passed by reference. This allows the function to change not just the contents of the array that \( da \) points to but also the pointer itself (so it can be made to point to the new array). Note how the function takes care not to go out of bounds in either array.

So we now know how to create arrays whose size is determined at run time. These array can also be resized as needed. You may now wonder why it is that C++ arrays are not all dynamically allocated (as in some other languages). The reason comes from our discussion of data management earlier in this chapter. Accessing an ordinary array only requires finding it in the activation record of a function, or among the data members of an object. In contrast, accessing a dynamically allocated array requires first accessing the pointer that points to the array and then using that pointer to access the array. The
void resize( int * & da, int old_size, int new_size )
// Changes the size of the dynamically allocated array of ints
// that da points to.
{
    int * da_new = new int[new_size];
    int min_size = ( old_size < new_size ? old_size : new_size );
    for ( int i = 0; i < min_size; ++i ) da_new[i] = da[i];

delete [] da;
da = da_new;
da_new = NULL;
}

Figure 6.4: A function that resizes a dynamically allocated array

pointer retrieval is an extra step that requires extra time. Therefore, ordinary
arrays typically lead to faster code. The gains are usually small but they can
be important in certain applications.

We end this section by reviewing all the operations we have learned that
either apply to pointers or are related to the dynamic allocation of arrays.
These operations are listed in Table 6.1.

Source code for the examples of this section is available on the course web
site under DynamicArrays.

Study Questions

6.3.1. What is the main disadvantage of dynamically allocated arrays?

6.3.2. How can a dynamically allocated array be resized?

6.3.3. In the function resize, could the argument da be declared as
        int * da?

6.3.4. What is a memory leak?

Exercises

6.3.5. Write a function
\*\*p
Declares a pointer of type T.

p = NULL
Makes p point to nothing.

p = q
Makes p point to where q points to.

p = new T[n]
Makes p point to a new dynamically allocated array of type T and size n.

delete [] p
Deallocates the array that p points to.

<table>
<thead>
<tr>
<th>Table 6.1: Some operations related to pointers and the dynamic allocation of arrays</th>
</tr>
</thead>
</table>

\*\*int\* copy(const int\* a, int n)

that takes as arguments an array a and its size n and returns a pointer to a new dynamically allocated array that contains a copy of all the elements of a.

6.3.6. Write a function

\*\*int\* concatenate(const int\* a, int n, const int\* b, int m)

that takes as arguments two arrays a and b and returns a pointer to a new dynamically allocated array that contains a copy of all the elements of a followed by a copy of all the elements of b. The arguments n and m are the sizes of the arrays a and b, respectively.

6.3.7. Consider the following arrays:

```c
int a[5];
int b[5];
```
What does each of the following statements do?

a) \( a = b; \)

b) \( a = c; \)

c) \( c = b; \)

d) \( c = d; \)

Verify your answers by running some tests.
Chapter 7

Implementation of Vectors

In this chapter, we will learn how to implement a basic class of vectors. In the process, we will also learn the important concept of generic programming.

7.1 Generic Programming

In the previous chapter, we wrote a function that resizes a dynamically allocated array (see Figure 6.4). As written, that function only resizes arrays of integers. What if we needed to resize arrays of strings or times?

One idea would be to copy the integer version of the function and replace every occurrence of \texttt{int} by \texttt{string} or \texttt{Time}. But we have to be careful because it may not be that every occurrence of \texttt{int} in the function refers to the type of element stored in the array. In fact, the index \texttt{i} that controls the loop should remain an integer in every version of \texttt{resize}.

A solution to this problem is to create a function template. In this template, the type of element stored in the array would be given a generic name, such as \texttt{T}, as shown in Figure 7.1. (Ignore the template declaration on the first line for now.) To get a function that resizes a particular type of array, all we need to do is copy the template and replace every \texttt{T} by the desired type.

This looks like a mechanical process that can be automated and, in fact, C++ compilers can do just that. This requires that we explicitly declare that the function is a template and that we identify the template arguments (or parameters). This is the purpose of the declaration on the first line of Figure 7.1.\footnote{That declaration uses the keyword \texttt{class} to declare \texttt{T}. This doesn’t mean that \texttt{T} must...}
template <class T>
void resize( T * & da, int old_size, int new_size )
// Changes the size of the dynamically allocated array that da
// points to. T is the type of element contained in the array.
// Assumption on T: has a default constructor.
{
    T * da_new = new T[new_size];
    int min_size = ( old_size < new_size ? old_size : new_size );
    for ( int i = 0; i < min_size; ++i ) da_new[i] = da[i];

delete [] da;
da = da_new;
da_new = NULL;
}

Figure 7.1: A template for the function resize

A function template is not really a function. It is used by the compiler
to generate functions as needed. This happens when the compiler sees a call
to the function. For example, if resize is called on an array of integers,
the compiler will look for a resize function that can take such an array as
argument. When that fails, the compiler will look for a template that can be
used to generate such a function. The above template will work as long as T
is set to int. The process of generating a function out of a template is called
template instantiation.

Even though a function template is not a function, we can think of it as a
generic function, that is, a function that can work on more than one type of
argument. This is an example of generic programming, the writing of code
that can be used on a variety of data types. We will soon learn that classes,
too, can be generic.

The fact that resize is a generic function does not necessarily mean that
it can be used on arrays of any type. In fact, if ever T is a class, when resize
creates an array of the new size, the default constructor of class T will be called
automatically to initialize each element of the array. This requires that the
class T include a default constructor. It is important to clearly document such

be a class. In this context, class simply means type. In fact, we can equivalently declare T
as a typename, which is more accurate. But the use of the word class is widespread.
7.1. GENERIC PROGRAMMING

```c++
template <class T>
void init( T * a, int n )
// Initializes an array to contain 0, 1, 2, ..., n-1. a points
// to the array, n is the size of the array and T is the type of
// element contained in the array.
// Assumption on T: the integers 0 to n-1 can be assigned to
// variables of type T.
{
    for ( int i = 0; i < n; ++i ) a[i] = i;
}

template <class T>
void print( const T * a, int n )
// Prints the contents of the array that a points to. n is the
// size of the array and T is the type of element contained in
// the array.
// Assumption on T: values of type T can be printed by using the
// output operator (<<).
{
    for ( int i = 0; i < n; ++i ) cout << a[i] << ' ';     
    cout << endl;
}
```

Figure 7.2: Templates for the functions `init` and `print`

conditions on template arguments, as shown in Figure 7.1.

Two additional examples of generic functions are given in Figure 7.2. Source code for the examples of this section is available on the course web site as `generic.cpp`.

Study Questions

7.1.1. What is a generic function?

7.1.2. What C++ construct allows us to implement generic functions?
Exercises

7.1.3. Write the following generic functions. In each case, the argument \( n \) is the size of the array. Document any necessary restrictions on the element type.

a) A generic function \( \text{max}(a, n) \) that returns the value of the maximum element of array \( a \).

b) A generic function \( \text{sum}(a, n) \) that returns the sum of the elements of array \( a \).

c) A generic function \( \text{count}(a, n, e) \) that returns the number of occurrences of element \( e \) in array \( a \).

7.2 A Basic Class of Vectors

Figure 7.3 shows the declaration of a basic class of vectors. (Additional methods and operators will be added in the following sections. You will be asked to add others in the exercises.)

The first thing to notice is that \( \text{Vector} \) is actually a class template. This is because we want to create a generic class of vectors, that is, vectors that can store any type of element. That’s what the template argument \( T \) represents: the type of element stored in the vector.

With function templates, the compiler is normally able to instantiate the template on its own. For example, as explained in Section 7.1, when \( \text{resize} \) is called on an array of integers, the compiler will realize that it needs to set the template argument \( T \) to \( \text{int} \). It’s the only way that the function call can be made to match the template.

But with a class template, the compiler needs our help. That’s because a declaration such as

\[
\text{Vector } v;
\]

would not allow the compiler to figure out the type of element we want to store in the vector. So we need to specify this type as follows:

\[
\text{Vector<}\text{int}\rangle v;
\]

This gives us a vector of integers. To get a vector of another type, simply replace \( \text{int} \) by that other type as in
template <class T>
class Vector
{
public:
    Vector() { buffer_ = NULL; size_ = 0; }
    explicit Vector( int n ) {
        buffer_ = create_new_buffer(n);
        size_ = n;
    }

    int size() const { return size_; }

    T & operator[]( int i ) { return buffer_[i]; }
    const T & operator[]( int i ) const { return buffer_[i]; }

    void resize( int n );

private:
    T * buffer_; // points to a dynamically allocated array that contains the vector elements
    int size_; // the number of elements in the vector (and the size of the buffer)

    // Returns pointer to new buffer of size n. Returns NULL if n <= 0.
    T * create_new_buffer( int n ) const {
        return (n > 0 ? new T[n] : NULL);
    }
};

Figure 7.3: The class declaration
Vector<string> v;

The class `Vector` has two data members. The first one is a pointer to a dynamically allocated array that contains the vector elements. The name `buffer` is standard for such an array. Recall that `buffer` is actually a pointer to the first element of that array. The type of this element is, of course, `T`. The second data member is the size of the vector, that is, the number of elements it holds.

We now turn to the constructors, methods and operators of the class. Most of them are implemented in the class declaration (see Figure 7.3). The method `size` simply returns the size of the vector.

The implementation of the indexing operator is straightforward but note that two versions are provided. The non-constant version returns a plain reference to the element. This reference allows the element not only to be retrieved, as in

```cpp
x = v[4];
cout << v[4];
```

but also modified, as in

```cpp
v[4] = 17;
cin >> v[4];
```

The constant version, on the other hand, returns a constant reference, one that does not allow the element to be modified.

Note that an alternative would be for the constant version to return a copy of the element:

```cpp
T operator[](int i) const { return buffer[i]; }
```

But since we don’t know how large elements of type `T` might be, it is safer to return a constant reference to avoid unnecessary copying.

The default constructor initializes the vector to be empty. The second constructor allows the creation of nonempty vectors. The argument is the initial size of the vector. The constructor uses the private method `create_new_buffer`. This is to ensure that the buffer is properly initialized to `NULL` in case the specified size is 0. (With some compilers, `new` may not return `NULL` when the size is 0.)
template <class T>
void Vector<T>::resize( int new_size )
{
    T * new_buffer = create_new_buffer(new_size);

    // copy elements to new buffer
    int min_size = ( size_ < new_size ? size_ : new_size );
    for ( int i = 0; i < min_size; ++i )
        new_buffer[i] = buffer_[i];

    // deallocate old buffer, if there is one
    if ( buffer_ != NULL ) delete [] buffer_;

    // give new buffer to receiver
    buffer_ = new_buffer;
    size_ = new_size;
}

Figure 7.4: The resize method

Note that it is better for create_new_buffer to be a private method instead of a public one. In part because that method is of no use to users of vectors. But also because this method is really part of the implementation of the class and it is therefore better to hide it from the users.

The second constructor has only one argument. Therefore, the compiler would normally use that constructor to perform implicit conversions. But the argument is an integer which implies that a statement such as \( v = 2 \) would cause the compiler to quietly convert the integer 2 into a vector of size 2. In this case, the conversion doesn’t seem to make much sense. So it is better to declare the constructor to be explicit. This prevents the constructor from being used in implicit conversions.

Figure 7.4 shows the implementation of the resize method. Note how we are careful to not go out of bounds in either buffer when copying the elements. In addition, the current buffer is not deallocated until after the elements have been copied to the new buffer. We also make sure there actually is a buffer to deallocate. Not doing so can cause the method to crash.

Our class Vector is a class template. Like any other class, we normally split it into two source files: a header file Vector.h that contains the class
declaration and an implementation file Vector.cpp that contains the implementation of all the methods and functions that weren’t already implemented in the header file. The header file is then included in any source file that contains client code, that is, code that uses the class (a test driver, for example). The class and the client code can then be compiled separately.

With class templates, however, the methods that are implemented in the implementation file will be instantiated only when the compiler compiles the code that uses the class. So the compiler needs the implementation file when it compiles the client code. In addition, there is usually no point in compiling the class template on its own since the compiler cannot instantiate it without seeing the client code.

Therefore, the usual way of organizing a class template is not to include the header file in the implementation file but to instead include the implementation file at the end of the header file and to instruct the compiler or IDE not to compile the implementation file. For example, in the case of our class template Vector, we would include Vector.h in every source file that contains client code but not in Vector.cpp. Instead, Vector.cpp is included at the end of Vector.h. The only files that are compiled are the implementation files containing client code (such as test.cpp). This is how the source code available on the course web site is organized.\(^2\)

Source code and a test driver for our class Vector are available on the course web site under Vector1.0.

**Study Questions**

7.2.1. Why is create_new_buffer declared private?

7.2.2. Why does a class like Vector need two versions of the indexing operator?

7.2.3. Why was the second constructor declared to be explicit?

7.2.4. Why is a template implementation file normally included in the template header file?

---

\(^2\)With Code::Blocks, template implementation files, such as Vector.cpp, can be excluded from the compilation by changing the properties of that file. Open the project in Code::Blocks, right-click on the name of the file, select Properties, select the Build tab and uncheck the boxes Compile file and Link file. The Code::Blocks project files available on the course web site are already set up that way.
7.3. **SOME ADDITIONAL METHODS**

**Exercises**

7.2.5. Add to `vector` a method `empty` that returns `true` if the vector is empty.

7.2.6. Add a constructor `vector(n, e)` that initializes the vector to contain `n` copies of element `e`.

7.2.7. Add a method `resize(n, e)` that pads the vector with copies of element `e` in case the size of the vector is increased. (Note that the original `resize(n)` can then be implemented as)

```cpp
void resize( int n ) { resize( n, T() ); }
```

since `T()` generates a default object of class `T`.

7.2.8. Add a method `clear` that makes the vector empty. Make sure that the buffer is properly deallocated.

7.2.9. Add equality and inequality testing operators (`==, !=`). Two vectors are equal if they are of the same size and contain the same element at each position.

7.2.10. Add a method `swap` that takes another vector as argument and exchanges their contents without copying any elements.

### 7.3 Some Additional Methods

In this section, we add three convenient methods to our class of vectors. All three work at the end, or back, of the vector.

The indexing operator allows us to access any element of a vector. But in many applications, the first and last elements of a vector are accessed more often than the others. Accessing the last element of a vector, for example, can of course be done by using the indexing operator as follows:

```cpp
v[ v.size() - 1 ]
```

The `back` method allows us to do this more easily as

```cpp
v.back()
```
template <class T>
class Vector
{
    public:
        T & back() { return buffer_[size_ - 1]; }
        const T & back() const { return buffer_[size_ - 1]; }

        void push_back(const T & new_element) {
            resize(size() + 1);
            back() = new_element;
        }

        void pop_back() { resize(size() - 1); }
    ... };

    Figure 7.5: The methods back, push_back and pop_back

The implementation of this method is straightforward, as can be seen in Figure 7.5.

Adding an element to the end of a vector is also a common operation in many applications. This can be done by using resize and back:

    v.resize( v.size() + 1 );
    v.back() = new_element;

The push_back method allows us to do this in one step:

    v.push_back( new_element );

The method can be implemented as shown in Figure 7.5.

Figure 7.5 also shows the implementation of pop_back, which removes the last element of the vector. Note how a simple resize does the job.

Updated source code and test driver are available on the course web site under Vector1.1.

Exercises

7.3.1. Add to vector methods front, push_front(e) and pop_front() that work just like back, push_back(e) and pop_back() but at the
7.4 Destroying and Copying Vectors

The current version of Vector has a couple of major flaws that have to do with the destruction and copying of vectors. We address these flaws in this section.

Suppose that a function contains a vector as a local variable. When the function returns, the Vector object ceases to exist but what happens to the buffer? Is it deallocated? The answer is no. Which implies that the memory space used by the buffer will continue to be reserved even though we have no longer any way of accessing that data. In other words, we will have a memory leak.

To recover that memory, we need to write a special method called a destructor. The name of a destructor is always the name of the class preceded by a tilde, as in ~Vector. Like constructors, the destructor has no return type. A Vector destructor can be declared and implemented as follows:

```cpp
~Vector() { if (buffer_ != NULL) delete [] buffer_; }
```

Note how we are careful to make sure that the vector does contain a buffer. Deallocation memory that a NULL pointer points to is an error that will normally cause a program to crash.

As a general rule, whenever a class contains pointers to dynamically allocated memory, we should consider whether we need to include a destructor. The key issue is whether the object that is being destroyed owns the dynamically allocated memory. If it does, then it is responsible for deallocating that memory. In the case of Vector, each Vector object owns its buffer.

There are circumstances where objects share access to dynamically allocated memory. In those cases, we need to decide which of these objects owns the memory and is responsible for eventually deallocating it.

We now turn to the copying of vectors. Every class contains a special constructor called the copy constructor. The role of the copy constructor, as its name implies, is to create copies of objects. The copy constructor is called in three different circumstances. First, when an object is created and initialized to be a copy of another one, through a declaration such as

```cpp
Vector<T> v1(v2);
```
which can also be written as

\[
\text{Vector}\langle \text{T} \rangle \ v1 = v2;
\]

Second, when an object is passed by value to a function as in

\[
\text{void } f( \text{Vector}\langle \text{T} \rangle \ v )
\]

Third, when a function returns an object, as in

\[
\text{Vector}\langle \text{T} \rangle \ g()
\{
  \text{Vector}\langle \text{T} \rangle \ v;
  \ldots
  \text{return } v;
\}
\]

(But note that some compilers are able to avoid copying return values.\textsuperscript{3})

Whenever we don’t write a copy constructor for a class, the compiler generates one automatically. But the compiler-generated copy constructor performs a shallow copy. As explained earlier, this means that the values of the data members are copied but if those data members are pointers, only the values of the pointers are copied, not what the pointers point to. In our case, this would result in two vectors that share the same buffer, which is not what we want.

Typically, when objects point to dynamically allocated memory, what we want is a deep copy. In our case, this means that we want the buffer to be copied so that each vector has its own copy of the buffer (and the elements it contains).

To get a copy constructor that performs a deep copy, we must write our own. An implementation of a Vector copy constructor is shown in Figure 7.6.

There is another method that the compiler automatically generates whenever we don’t write one: the assignment operator (\(=\)). As in the case of the copy constructor, the compiler-generated assignment operator performs a shallow copy. To get an assignment operator that performs a deep copy, we must write our own.

A possible assignment operator for Vector is shown in Figure 7.7. The

\footnote{This is done by essentially storing the local variable that is returned (\(v\) in this case) in the memory location where the return value would be copied to. This is an example of a compiler optimization. This particular one is called return value optimization.}
template <class T>
Vector<T>::Vector( const Vector & v ) {
    buffer_ = create_new_buffer( v.size() );
    for ( int i = 0; i < v.size(); ++i ) buffer_[i] = v[i];
    size_ = v.size();
}

Figure 7.6: Copy constructor

template <class T>
Vector<T> & Vector<T>::operator=( const Vector<T> & v ) {
    T * new_buffer = create_new_buffer( v.size() );
    for ( int i = 0; i < v.size(); ++i ) new_buffer[i] = v[i];

    // deallocate old buffer, if there is one
    if ( buffer_ != NULL ) delete [] buffer_;

    // give new buffer to receiver
    buffer_ = new_buffer;
    size_ = v.size();

    // return receiver
    return *this;
}

Figure 7.7: Assignment operator
template <class T>
Vector<T> & Vector<T>::operator=( const Vector<T> & v )
{
    resize( v.size() );
    for ( int i = 0; i < size(); ++i ) buffer[i] = v[i];

    // return receiver
    return *this;
}

Figure 7.8: A less efficient implementation of the assignment operator

operator has only one argument, which corresponds to the vector on the right hand side of the assignment. The vector on the left plays the role of receiver. In other words,

v1 = v2;

is understood by the compiler as

v1.operator=(v2);

Note that the assignment operator does not deallocate the current buffer of the receiver until a new buffer has been successfully allocated and filled with a copy of the argument’s buffer. This prevents the receiver from losing its buffer in case the allocation fails or in case the operator was used to do assign a vector to itself as in

v = v;

A simpler implementation of the operator is possible by using the resize method to resize the receiver to match the argument’s size, as shown in Figure 7.8. But this implementation is less efficient since it unnecessarily copies the current elements of the receiver during the resizing.

The return value of the assignment operator requires some explanation. The operator is supposed to return a reference to its receiver. This allows chains of assignments such as

v1 = v2 = v3;

which are essentially executed as
7.4. DESTROYING AND COPYING VECTORS

\[ v1.\text{operator} = ( v2.\text{operator} = (v3) ); \]

In other words, the return value of the second assignment serves as the argument (and right operand) of the first assignment.

Now, to return its receiver, the assignment operator uses the fact that within any method of any class, the variable \texttt{this} always points to the receiver. But we don’t want to return a pointer to the receiver; we need to return the receiver itself. This is accomplished by \textit{dereferencing} the pointer, which means accessing the value that the pointer points to. This is done by using the \textit{dereferencing operator} \( * \). In general, whenever \( p \) is an pointer, \( *p \) refers the value that the pointer points to. Therefore, the assignment operator returns \( *\texttt{this} \).

In conclusion, as a general rule, whenever we create a class that contains pointers to dynamically allocated memory, we should always consider whether we need to write our own destructor, copy constructor and assignment operator. We usually do.

The complete declaration of the class \texttt{Vector} is shown in Figure 7.9. It shows all the methods we implemented in this chapter. (The implementations of some of the inline methods have been removed to allow the class declaration to fit on one page.) The entire source code is available on the course web page under \texttt{Vector1.2}.

\textbf{Study Questions}

7.4.1. When is the destructor is called?

7.4.2. What are the three circumstances in which the copy constructor is called?

7.4.3. What kind of copy do the compiler-generated copy constructor and assignment operator perform?

7.4.4. Why doesn’t our implementation of the assignment operator begin by deallocating the receiver’s buffer?

7.4.5. What is the disadvantage of using \texttt{resize} in the implementation of the assignment operator?

7.4.6. Why does the assignment operator return its receiver?
template <class T>
class Vector
{
    public:
        Vector() { buffer_ = NULL; size_ = 0; }

        explicit Vector( int n ) { ... }

        Vector( const Vector & v );

    ~Vector() { if (buffer_ != NULL) delete [] buffer_; }

        int size() const { return size_; }

        T & operator[]( int i ) { return buffer_[i]; }
        const T & operator[]( int i ) const { return buffer_[i]; }

        void resize( int new_size );

        Vector<T> & operator=( const Vector<T> & v );

        T & back() { return buffer_[ size_ - 1 ]; }
        const T & back() const { return buffer_[ size_ - 1 ]; }

        void push_back( const T & new_element ) { ... }

        void pop_back() { resize( size() - 1 ); }

    private:
        T * buffer_;
        int size_;

    T * create_new_buffer( int n ) const { ... }
};

Figure 7.9: The complete class declaration
7.4. DESTROYING AND COPYING VECTORS

7.4.7. In the body of a method, what does \texttt{this} refer to?

7.4.8. What operator can be used to access the value that a pointer points to?

\textbf{Exercises}

7.4.9. Verify that the destructor is really called when an object ceases to exist. Do this by adding an output message to the destructor and by writing an appropriate test driver.

7.4.10. Verify that the copy constructor is really called in the three circumstances explained in this section. Do this by adding an output message to the copy constructor and writing an appropriate test driver. (Keep in mind the possibility that your compiler performs the return value optimization mentioned earlier. See if you can turn it off in your compiler’s settings.)

7.4.11. Add to \texttt{Vector} a method \texttt{assign(n, e)} that sets the vector to contain \texttt{n} copies of element \texttt{e}. Make sure that the buffer is properly deallocated and that vector elements are not copied unnecessarily.
Chapter 8

Analysis of Algorithms

In this chapter, we will learn how to analyze algorithms in order to evaluate their efficiency. We will see how analysis can be carried out from pseudocode, which allows us to choose efficient algorithms without having to implement the inefficient ones. We will also discuss the relative benefits and disadvantages of analysis as opposed to measuring exact running times through testing.

8.1 Introduction

From a user’s perspective, quality software must be reliable, robust, easy to use and efficient. In these notes, we have also emphasized that software, especially large software, must be designed so it is easy to understand, code, test and modify. And we have seen that modularity and abstraction help to achieve these qualities.

There is a lot more that can be learned about all of these topics. For example, reliability is normally achieved through a mix of testing and verification. At Clarkson, testing is covered in more detail in a course such as CS350 Software Design and Development while verification is the main focus of CS458 Formal Methods for Software Verification. As its title indicates, CS350 also covers software design. Usability is the main topic of CS459 Human-Computer Interaction, while efficiency is the central focus of CS344 Algorithms and Data Structures and CS447 Computer Algorithms.

Up until now, efficiency has not been a main focus of these notes. But efficiency will be a main consideration in the rest of these notes as we revisit the design of our phone book program, design a text editor, improve the efficiency
of our class of vectors and learn efficient algorithms for searching and sorting. Note that in one form or another, most software systems incorporate some amount of searching and sorting of data.

But first, we need to learn to evaluate the efficiency of algorithms. We will do this by analyzing the algorithms. We will learn that this can be done from pseudocode, which allows us to choose efficient algorithms without wasting time and energy implementing inefficient alternatives.

Note that algorithm analysis is also useful for the analysis of data structures, if only because data structure operations are algorithms.

In general, an algorithm is efficient if it uses a small amount of computational resources. The two resources that are most often considered are running time and memory space. An example of another resource is randomness.\(^1\) In this chapter, we will focus on running time but the main concepts and techniques we will learn also apply to other resources.

**Study Questions**

8.1.1. What are some of the properties quality software should have?

8.1.2. What does it mean for software to be efficient?

8.1.3. What two computational resources are most often considered?

8.2 **Measuring Exact Running Times**

When choosing or designing an algorithm for a particular problem, there are two questions that can be asked: Is the algorithm fast enough? Is it as fast as possible?

The first question is perhaps the more pragmatic. To be able to answer that question, however, we need to know exactly what is meant by fast enough. One possibility would be precise time targets such as 5 ms. Now, the running time of an algorithm depends on several factors including what data it is used on, what computer it runs on and exactly how it is coded. (The input data could be arguments, input files or data entered by the user.) If all that information is available, then tests can be run to accurately determine if the algorithm is fast enough.

\(^1\)Algorithms that use randomness are usually studied in a course such as CS447 *Computer Algorithms.*
8.2. MEASURING EXACT RUNNING TIMES

<table>
<thead>
<tr>
<th>$n$</th>
<th>10</th>
<th>$10^3$</th>
<th>$10^6$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\log_2 n \ \mu s$</td>
<td>3 $\mu s$</td>
<td>10 $\mu s$</td>
<td>20 $\mu s$</td>
</tr>
<tr>
<td>$n \ \mu s$</td>
<td>10 $\mu s$</td>
<td>1 ms</td>
<td>1 s</td>
</tr>
<tr>
<td>$n^2 \ \mu s$</td>
<td>100 $\mu s$</td>
<td>1 s</td>
<td>12 days</td>
</tr>
</tbody>
</table>

Table 8.1: Running times of three algorithms

But very often, there are no precise time targets to meet. In that case, the safest approach is to choose the fastest algorithm among the available alternatives. So how can we determine which of several possible algorithms is fastest?

An obvious way is to implement each of the algorithms, run them and measure their running times. The choice of what computer to use probably doesn’t matter much since if an algorithm is significantly faster than another on one computer, the same is probably true on most if not all computers.

A more delicate issue is what inputs to use for the tests. Very often, we need an algorithm that will run well on a wide variety of inputs. So we could run tests on various inputs and compute the average running time of each algorithm. But the running time of an algorithm can vary greatly, especially as a function of the size of the input.

For example, suppose that three algorithms have running times $\log n \ \mu s$, $n \ \mu s$ and $n^2 \ \mu s$, where $n$ is the size of the input. Table 8.1 shows what these running times are for various input sizes. When the input size is only 10, the difference between the running time of these three algorithms run is not that large. But at $n = 10^3$, the difference is significant and at $n = 10^6$, it is huge. Therefore, when comparing algorithms by measuring their running times, it is important to use a wide range of input sizes.

So we can determine which of several algorithms will be the fastest as follows: implement the algorithms, run them on a wide variety of inputs, and measure the running times. Of course, for the comparisons to be valid, the algorithms must be coded in the same language and run on the same computer and similar inputs.

This approach has several significant disadvantages. First, it requires that all the algorithms be implemented, even those than will end up not being used. Second, writing test drivers and running tests takes time, especially since we
for i = 0 to n-1
    print a[i]

Figure 8.1: Printing the contents of an array

must test on a good number of inputs of each size to make sure we have a representative sample. Third, because all the algorithms being compared must be implemented in the same language and tested on the same computer and on similar inputs, earlier tests done on different computers, with different inputs, or using different programming languages often need to be repeated.

In the rest of this chapter, we will learn that it is possible evaluate the running time of an algorithm in a way that addresses these problems.

Study Questions

8.2.1. When comparing the efficiency of algorithms, why is it usually important to compare running times over a wide range of input sizes?

8.2.2. What are three significant weaknesses of comparing algorithms by measuring exact running times?

8.3 Analysis

Our goal is to find a way to assess the running time of an algorithm without having to implement and test it. We also want this assessment to be valid for all implementations of the algorithm and for all the computers on which the algorithm may run. And, of course, to be useful, this assessment should allow us to compare the running time of various algorithms.

Let’s consider an example. Figure 8.1 shows pseudocode for an algorithm that prints the contents of an array. The running time of this algorithm can be determined as follows. Before the first iteration of the loop, i is initialized and its value is compared to n-1. At every iteration of the loop, an array element is accessed, then printed, i is incremented and then again compared to n. The loop is executed n times. Therefore, the running time of the algorithm is

\[ t(n) = c_{\text{assign}} + c_{\text{comp}} + (c_{\text{index}} + c_{\text{print}} + c_{\text{incr}} + c_{\text{comp}})n \]
where the $c$ constants are the running times of the various basic operations performed by the algorithm. For example, $c_{\text{assign}}$ is the time it takes to assign a value to an integer variable.\(^2\)

We can simplify this expression by letting $a = c_{\text{index}} + c_{\text{print}} + c_{\text{incr}} + c_{\text{comp}}$ and $b = c_{\text{assign}} + c_{\text{comp}}$. The running time of the algorithm can then be written as

$$t(n) = an + b$$

If we knew the exact values of the constants $a$ and $b$, this expression would allow us to determine the exact running time of the algorithm on inputs of any size. But the values of these constants depend on exactly how the algorithm is implemented and on which computer the algorithm will run. Recall that we want to assess the running time of an algorithm without having to implement it. We also want this assessment to be valid for all computers. Therefore, we will not determine the values of the constants and instead focus on the “general form” of the running time as a function of $n$.

In our example, the running time of the printing algorithm is a linear function of $n$. Is that useful information? Knowing that the running time is a linear function doesn’t allow us to determine the exact running time of the algorithm for any input size. But suppose that another algorithm has a running time that’s a quadratic function of $n$, for example. Then we know that when $n$ is large enough, the printing algorithm runs faster, much faster, than this other algorithm. This basic fact about linear and quadratic functions is apparent in the numbers that were given in Table 8.1. Therefore, is is useful to know that the running time of the printing algorithm is a linear function of $n$.

So analyzing an algorithm to determine the general form of its running time is a useful alternative to the measurement of exact running times through testing. It is useful because it can be used to determine that an algorithm will be faster than another one on every input that is large enough.

\(^2\)It is not exactly true that the running time of these basic operations is constant. For example, the time it takes to assign a value to an integer variable usually depends on the maximum value that can be held in that integer variable. But it is common practice to consider that all the basic operations typically provided by programming languages can be executed in constant time. A more precise analysis would rarely lead to different conclusions. This issue is normally examined in more detail in courses such as CS344 *Algorithms and Data Structures* and CS447 *Computer Algorithms*. The representation of integer values in a computer’s memory, as well as some aspects of the implementation of the basic operations on those integers, are covered in CS241 *Computer Organization*. 
Analysis has three main advantages over measuring exact running time through testing. First, analysis can be carried out from pseudocode, without having to implement the algorithms. Second, analysis does not require writing test drivers or performing possibly time-consuming tests. Third, each algorithm needs to be analyzed only once because the results of the analysis are valid for every (reasonable) implementation of the algorithm and every computer and data the algorithm may run on.

On the other hand, analysis has two main disadvantages over measuring exact running times. First, it is not as precise. For example, it does not allow us to distinguish between two linear-time algorithms or to determine if an algorithm meets specific time targets. Second, analysis is valid only for large enough inputs, not for small ones.

In general, analysis is a convenient and reliable way of quickly identifying large differences in running times. When more accuracy is needed, or when the analysis is too difficult, which can happen, we must then resort to measuring exact running times through testing.

Study Questions

8.3.1. As described in this section, what does analysis seek to determine?

8.3.2. What are three advantages and two disadvantages of analysis over the measurement of exact running times through testing?

8.4 Asymptotic Running Times

In the previous section, we saw that the general form of the running time of an algorithm, when expressed as a function of a parameter such as its input size, is a useful measure of the efficiency of the algorithm. For example, if we determine that an algorithm has a linear running time then we know that it will run faster than any quadratic-time algorithm on every input that is large enough.

But what should we make of a running time of the form $an + b \log n + c$? How does that compare to linear and quadratic running times, for example?

The key point to remember is that asymptotic analysis allows us to compare the running time of algorithms for large enough input sizes. When $n$ is large enough, the terms $b \log n$ and $c$ are insignificant compared to $an$. In other
words, the dominant term \( an \) is the one that will essentially determine the running time for large enough values of \( n \). This means that when \( n \) is large enough, \( an + b \log n + c \) will behave essentially like the linear function \( an \). In particular, \( an + b \log n + c \) will be much smaller than any quadratic function for every large enough \( n \).

Therefore, when doing asymptotic analysis, we can focus on determining the dominant term of the running time. In addition, since the value of the constant factor of the dominant term is not known (because it depends on a particular implementation and computer), it is irrelevant to our analysis and we might as well ignore it. What we are left with is what can be called the *asymptotic running time* of an algorithm. For example, if the running time of an algorithm is \( an + b \log n + c \), then we say that its asymptotic running time is \( n \).

The relationship between the exact running time of an algorithm and its asymptotic running time can be made precise through the notion of *asymptotic equivalence*. We say that two running times are asymptotically equivalent if they are within a constant factor of each other for every large enough input. This can be formalized as follows:

**Definition 8.1** We say that \( f(n) \) is asymptotically equivalent to \( g(n) \), or that \( f(n) \) is \( \Theta(g(n)) \) ("\( f(n) \) is Theta of \( g(n) \)") if there are positive constants \( a, b \) and \( n_0 \) such that for every \( n \geq n_0 \),

\[
ag(n) \leq f(n) \leq bg(n)
\]

Loosely speaking, when a function is asymptotically equivalent to another one, it means that when \( n \) is large enough, the two functions have similar values. So we can view asymptotic equivalence as meaning “about the same”.

The relationship between the exact running time of an algorithm and its asymptotic running time is given by the following property of asymptotic equivalence:

**Property 8.2** If \( f(n) \) has dominant term \( cg(n) \), where \( c \) is a constant, then \( f(n) \) is \( \Theta(g(n)) \).

In our example, we have that the dominant term of \( an + b \log n + c \) is \( an \). Therefore, \( an + b \log n + c \) is \( \Theta(n) \).

Note that it is also true that \( an + b \log n + c \) is \( \Theta(an) \) and that \( an + b \log n + c \) is \( \Theta(an + b \log n + c) \). But the statement \( an + b \log n + c \) is \( \Theta(n) \) is more useful
because it makes it easier to compare to other running times. For example, every running time of form \(an+b\) is also \(\Theta(n)\). This implies that \(an+b \log n + c\) is asymptotically equivalent to every running time that’s a linear function of \(n\).

The statement “\(f(n) = \Theta(g(n))\)” is often written “\(f(n) = \Theta(g(n))\).” But note that this equal sign is not a real equal sign. In particular, it doesn’t make sense to write “\(\Theta(g(n)) = f(n)\).”

We will not define precisely what is meant by dominant term. This could be done but involves mathematical concepts that some of you may not be familiar with.\(^3\) In most situations, it is clear what the dominant term of a running time is.

In the previous section, we said that the goal of analysis is to determine the “general form” of the running time of an algorithm. We can now be more precise: the goal is to determine the asymptotic running time of an algorithm. For this reason, this type of analysis is called asymptotic analysis.

In other words, when we do asymptotic analysis, we determine the running time of an algorithm as a function of its input size (or some other parameter), we simplify the running time by keeping only the dominant term and removing its constant factor, and we use the \(\Theta\) notation. In addition, when we say that an algorithm has asymptotic running time \(f(n)\), what we mean is that the running time of the algorithm is \(\Theta(f(n))\) and that we have removed low-order terms and constant factors from \(f(n)\).

Several examples of asymptotic running times, and how they compare to each other, will be given in the next section. Later in this chapter, we will learn basic strategies for analyzing the running time of simple algorithms.

**Study Questions**

8.4.1. What is the asymptotic running time of an algorithm?

8.4.2. How exactly does the asymptotic running time of an algorithm relate to its exact running time?

8.4.3. What does it mean for two running times to be asymptotically equivalent?

\(^3\)One possible definition is to say that \(g(n)\) is the dominant term of \(f(n)\) if for every other term \(h(n)\) of \(f(n)\), we have that \(\lim_{n \to \infty} (h(n)/g(n)) = 0\).
8.4.4. What is the main advantage of simplifying the running time of an algorithm?

8.4.5. What is asymptotic analysis?

**Exercises**

8.4.6. Below are a series of statements of the form \( f(n) = \Theta(g(n)) \). Prove that each of these statements is correct by finding, in each case, positive constants \( a, b \) and \( n_0 \) such that \( ag(n) \leq f(n) \leq bg(n) \) for every \( n \geq n_0 \). Justify your answers.

   a) \( n + 10 = \Theta(n) \).
   b) \( n^2 + n = \Theta(n^2) \).
   c) \( 3n^2 - n = \Theta(n^2) \).
   d) \( 3n^2 - n + 10 = \Theta(n^2) \).

8.4.7. Show that if \( c \) and \( d \) are any two numbers greater than 1, then \( \log_c n = \Theta(\log_d n) \). (This implies that when specifying running times using the \( \Theta \) notation, it is not necessary to specify the base of logarithms.)

8.5 Some Common Running Times

Table 8.2 gives a list of common asymptotic running times, in order, from smallest to largest. In this table, \( c \) represents a constant. The running times in Table 8.2 are listed in increasing order in the sense that when \( n \) is sufficiently large, each running time in this table is much larger than the preceding ones and much smaller than the following ones. (You need \( k > 2 \) for \( n^k \) to be larger than \( n^2 \).)

We already saw numbers that show how large a difference there is between logarithmic, linear and quadratic running times (see Table 8.1). Table 8.3 provides some numbers that compare linear, quadratic and exponential running times. These tables make it clear that quadratic-time algorithms are usually impractical on large inputs and that exponential-time algorithms are useless even for inputs of a moderate size.
### Running Time

<table>
<thead>
<tr>
<th>Common Name</th>
<th>Typical Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>constant</td>
<td>a single basic operation</td>
</tr>
<tr>
<td>logarithmic</td>
<td>fast searching algorithms</td>
</tr>
<tr>
<td>linear</td>
<td>simple searching algorithms</td>
</tr>
<tr>
<td>(n \log n)</td>
<td>fast sorting algorithms</td>
</tr>
<tr>
<td>(n^2)</td>
<td>simple sorting algorithms</td>
</tr>
<tr>
<td>polynomial</td>
<td>most algorithms that are fast enough to be useful in practice</td>
</tr>
<tr>
<td>exponential</td>
<td>exhaustive searches of very large sets</td>
</tr>
<tr>
<td>factorial</td>
<td>same as above</td>
</tr>
</tbody>
</table>

Table 8.2: Some common running times

<table>
<thead>
<tr>
<th>(n)</th>
<th>(10)</th>
<th>(20)</th>
<th>(40)</th>
<th>(60)</th>
<th>(80)</th>
</tr>
</thead>
<tbody>
<tr>
<td>(n \mu s)</td>
<td>10 (\mu s)</td>
<td>20 (\mu s)</td>
<td>40 (\mu s)</td>
<td>60 (\mu s)</td>
<td>80 (\mu s)</td>
</tr>
<tr>
<td>(n^2 \mu s)</td>
<td>0.1 ms</td>
<td>0.4 ms</td>
<td>1.6 ms</td>
<td>3.6 ms</td>
<td>6.4 ms</td>
</tr>
<tr>
<td>(2^n \mu s)</td>
<td>1 ms</td>
<td>1 s</td>
<td>13 days</td>
<td>(37 \times 10^3) years</td>
<td>(38 \times 10^9) years</td>
</tr>
</tbody>
</table>

Table 8.3: More execution times
Exercises

8.5.1. To each of Tables 8.1 and 8.3, add rows for the running times $n \log_2 n$ and $n^3$.

8.5.2. How large does $n$ have to be before $2^n$ is larger than each of the following functions: $n$, $n^2$, $n^3$ and $n^6$?

8.6 Basic Strategies

The examples in this section illustrate basic strategies that can be used in the analysis of simple algorithms. Some of these strategies rely on certain properties of asymptotic equivalence (the $\Theta$ notation).

**Example 8.3** Many algorithms are a sequence of steps performed by other algorithms. Suppose that an algorithm consists of three steps, performed by algorithms $A$, $B$ and $C$, in that order. Let $T_A(n)$, $T_B(n)$ and $T_C(n)$ denote the running time of these algorithms. Then the running time of the overall algorithm is simply the sum of those running times:

$$T(n) = T_A(n) + T_B(n) + T_C(n)$$

Now suppose that the running times of these algorithms are $\Theta(n)$, $\Theta(1)$ and $\Theta(n)$, respectively. Then we can write that

$$T(n) = \Theta(n) + \Theta(1) + \Theta(n)$$

It should be clear that one of the $\Theta(n)$ functions will dominate and therefore that $T(n)$ is $\Theta(n)$. This conclusion can be justified by the following property of asymptotic equivalence:

**Property 8.4 (Additivity)** Suppose that $f_1(n) = \Theta(g_1(n))$ and $f_2(n) = \Theta(g_2(n))$. Then $f_1(n) + f_2(n) = \Theta(g_1(n) + g_2(n))$.

Intuitively, this property makes sense: if $f_1(n)$ is about the same as $g_1(n)$, and if $f_2(n)$ is about the same $g_2(n)$, then $f_1(n) + f_2(n)$ should be about the same as $g_1(n) + g_2(n)$.

In our case, we have that $T_A(n) = \Theta(n)$ and $T_B(n) = \Theta(1)$. Therefore, $T_A(n) + T_B(n) = \Theta(n + 1)$. We also have that $T_C(n) = \Theta(n)$. Therefore,

$$T(n) = T_A(n) + T_B(n) + T_C(n) = \Theta(n + 1 + n) = \Theta(2n + 1)$$
Now, $2n + 1 = \Theta(n)$. This should imply that $T(n) = \Theta(n)$. This is confirmed by another useful property of asymptotic equivalence:

**Property 8.5 (Transitivity)** If $f(n)$ is $\Theta(g(n))$ and $g(n)$ is $\Theta(h(n))$, then $f(n)$ is $\Theta(h(n))$.

Once again, transitivity makes sense: if $f(n)$ is about the same as $g(n)$ and $g(n)$ is about the same as $h(n)$, then $f(n)$ should be about the same as $h(n)$.

To summarize,

$$T(n) = \Theta(n) + \Theta(1) + \Theta(n)$$

$$= \Theta(n + 1 + n) \text{ (by additivity)}$$

$$= \Theta(2n + 1)$$

$$= \Theta(n) \text{ (since } 2n + 1 = \Theta(n) \text{ and by transitivity)}$$

\[\square\]

**Example 8.6** To practice, let’s do a variation on the previous example. Suppose that the running times of the algorithms $A$, $B$ and $C$ are now $\Theta(n)$, $\Theta(n^2)$ and $\Theta(1)$, respectively. Then

$$T(n) = \Theta(n) + \Theta(n^2) + \Theta(1)$$

$$= \Theta(n + n^2 + 1)$$

$$= \Theta(n^2)$$

\[\square\]

**Example 8.7** Besides consecutive steps, loops are another very common form of algorithm. Figure 8.2 shows an algorithm we considered earlier in this chapter. It prints the contents of an array. This algorithm has the very common general form

```
for (init; test; update) body
```

where `init` stands for initialization. In our case, the initialization is $i = 0$, the test could be $i < n$, the update is $++i$ and the body is print $a[i]$.

Very frequently, as is the case here, the operations that manage the loop all run in constant time and the body of the loop takes the same amount of
for i = 0 to n-1
    print a[i]

Figure 8.2: Printing the contents of an array

for i = 0 to n-1
    for j = 0 to n-1
        print a[i,j]

Figure 8.3: Printing the contents of a two-dimensional array

time at each iteration of the loop. In this example, the body of the loop also
runs in constant time. Therefore, the running time of the loop is

$$T(n) = c_{\text{init}} + c_{\text{test}} + n(c_{\text{body}} + c_{\text{update}} + c_{\text{test}})$$

The dominant term in this expression is $nc_{\text{body}}$. Therefore, $T(n) = \Theta(n)$.

Example 8.8 Figure 8.3 shows an algorithm that consists of two nested loops.
It prints the contents of a two-dimensional array. The usual strategy for ana-
lyzing nested loops is to work from the inside out.

The inner loop can be analyzed as in the previous example. Its running
time is

$$T_{\text{inner}}(n) = \Theta(n)$$

The outer loop is also of the general form

for (init; test; update) body

But this time, the body of the loop (which is the inner loop) does not run in
constant time. Therefore, the running time of the outer loop is

$$T_{\text{outer}}(n) = c_{\text{init}} + c_{\text{test}} + n(t_{\text{body}}(n) + c_{\text{update}} + c_{\text{test}})$$

The dominant term in this expression is $nt_{\text{body}}(n)$. Therefore,

$$T_{\text{outer}}(n) = \Theta(nt_{\text{body}}(n))$$

Now, $t_{\text{body}}(n) = \Theta(n)$ since the body of the outer loop is the inner loop. It
seems clear that this implies that $nt_{\text{body}}(n) = \Theta(n^2)$. This conclusion can be
justified by another property of asymptotic equivalence:
for $i = 0$ to $n-1$
    for $j = 0$ to $i-1$
        print $a[i,j]$

Figure 8.4: Printing the lower left triangle of a two-dimensional array

**Property 8.9 (Multiplicativity)** Suppose that $f_1(n) = \Theta(g_1(n))$ and $f_2(n) = \Theta(g_2(n))$. Then $f_1(n)f_2(n) = \Theta(g_1(n)g_2(n))$.

In our case, we have that $n = \Theta(n)$ and $t_{\text{body}}(n) = \Theta(n)$. Therefore, $nt_{\text{body}}(n) = \Theta(n^2)$. Since $T_{\text{outer}}(n) = \Theta(nt_{\text{body}}(n))$, we get that $T_{\text{outer}}(n) = \Theta(n^2)$, by transitivity.  

**Example 8.10** Let’s analyze a slightly more complicated loop. Figure 8.4 shows an algorithm that prints the lower left triangle of a two-dimensional array.

Once again, we analyze the inner loop first. The only difference compared to the previous example, is that the loop iterates $i$ times. This implies that the running time of the inner loop is of the form $ai + b$, which means that the running time of the inner loop varies with $i$. This has the important consequence that we cannot simply multiply the running time of the inner loop by the number of times it executes. Instead, we need to add the running time of all the executions of the inner loop.

Ignoring the constant-time operations that manage the outer loop, we get that

$$T_{\text{outer}}(n) = \Theta\left(\sum_{i=0}^{n-1}(ai + b)\right)$$
if \( n < 10 \)
    sort using simple sorting algorithm
else
    sort using fast sorting algorithm

Figure 8.5: A hybrid sorting algorithm

Now,
\[
\sum_{i=0}^{n-1} (a_i + b) = \sum_{i=0}^{n-1} a_i + \sum_{i=0}^{n-1} b
\]
\[
= a \sum_{i=0}^{n-1} i + bn
\]
\[
= \frac{(n - 1)n}{2} + bn
\]
\[
= \frac{a}{2}n^2 - \frac{a}{2}n + bn
\]
\[
= \Theta(n^2)
\]

Therefore, the running time of the outer loop is \( \Theta(n^2) \).

Example 8.11 Later in these notes, we will learn that there are simple sorting algorithms that run in time \( \Theta(n^2) \) and more complex sorting algorithms that run much faster, in time \( \Theta(n \log n) \). On small inputs, however, the simple sorting algorithms often run faster than the more complex ones. Figure 8.5 shows a hybrid sorting algorithm that takes advantage of that fact.

Now, what is the overall running time of this algorithm, \( \Theta(n^2) \) or \( \Theta(n \log n) \)? The important thing to remember is that the asymptotic running time of an algorithm is determined by its running time on large inputs. The fast sorting algorithm is used for all \( n \geq 10 \). Therefore, the asymptotic running time of the hybrid algorithm is the running time of the fast sorting algorithm, which is \( \Theta(n \log n) \).

We end this section with two additional properties of asymptotic equivalence:

Property 8.12 (Reflexivity) Every function is asymptotically equivalent to itself. That is, every function \( f(n) \) is \( \Theta(f(n)) \).
Property 8.13 (Symmetry) If $f(n)$ is $\Theta(g(n))$, then $g(n)$ is $\Theta(f(n))$.

When a relation is reflexive, symmetric and transitive, as is the case with asymptotic equivalence, we say that it is an equivalence relation. This important mathematical concept is normally studied in a course such as MA211 Foundations.

Exercises

8.6.1. What is the (asymptotic) running time of each of the following algorithms, as a function of $n$? Don’t forget to simplify and use the $\Theta$ notation. Justify your answers.

a) 
   ```
   for i = 1 to n
       for j = 1 to 2n+1
           print '∗'
   ```

b) 
   ```
   for i = 1 to 10
       for j = 1 to n
           print '∗'
   ```

c) 
   ```
   for i = 1 to n
       for j = i to i+5
           print '∗'
   ```

d) 
   ```
   for i = 1 to n
       for j = i to n
           print '∗'
   ```

e) 
   ```
   for i = 1 to n
       for j = 1 to 2*i+1
           print '∗'
   ```

f) 
   ```
   for i = 1 to n*n
       for j = 1 to i
           print '∗'
   ```
8.7. WORST-CASE AND AVERAGE-CASE ANALYSIS

8.7.1. Consider the sequential search algorithm shown in Figure 8.7. What is the running time of this algorithm? The accurate answer is that it depends on the location of the first occurrence of \( x \) in the array.

We can talk of at least three different running times for a given algorithm. All are functions of the input size. The best-case running time is the minimum...
CHAPTER 8. ANALYSIS OF ALGORITHMS

running time required on inputs of size $n$. In the case of the sequential search algorithm, the best case occurs when $x$ is the first element of the array. In that case, the running time is constant.

The worst-case running time is the maximum running time required on inputs of size $n$. In our example, the worst case occurs when $x$ is not found. In that case, the running time is linear in $n$.

The average-case running time is the average running time required on inputs of size $n$. This running time is usually more difficult to determine, in part because it requires knowing how likely each input of size $n$ is. For example, for the sequential search, how likely is it that $x$ will not be found? Given that it is found, how likely is it that it will be found in each of the possible positions?

In this example, one possible approach is to determine the average-case running time for the two separate cases of a successful and an unsuccessful search. If the search is unsuccessful, the running time will always be the same, so the average and worst-case running times are the same: $\Theta(n)$.

In the case of a successful search, a common approach when lacking any more precise knowledge of the particular application we have in mind, is to assume that each location is equally likely. It is easy to see that the running time of the search is of the form $ak + b$, where $k$ is the index of the first occurrence of $x$. Taking the average over all possible positions, we get

$$\frac{1}{n} \sum_{k=0}^{n-1} (ak + b) = \frac{1}{n} \left( a \sum_{k=0}^{n-1} k + bn \right)$$

$$= \frac{a}{n} \sum_{k=0}^{n-1} k + b$$

$$= \frac{a}{n} \frac{(n-1)n}{2} + b$$

$$= \frac{a}{2} \frac{n-1}{2} + b$$

$$= \Theta(n)$$

Therefore, the average running time of a successful search is also $\Theta(n)$.

In general, the best-case running time is not very useful. The worst-case running time is much more useful and has the advantage of giving us a guarantee because it is an upper bound on the running time required for all inputs (that are large enough). A possible disadvantage of the worst-case running
time is that this upper bound may be much larger than the running time re-
quired by most inputs. In other words, the worst-case running time can be
overly pessimistic.

An example of this occurs with the quicksort algorithm, one of the fast
sorting algorithms we will study later in these notes. This algorithm has
a worst-case running time of $\Theta(n^2)$ while the mergesort algorithm, another
fast sorting algorithm, has a $\Theta(n \log n)$ worst-case running time. This might
indicate that quicksort is much slower than mergesort. However, in practice,
quicksort usually runs faster than mergesort.

In part, this can be explained by the fact that the average-case running time
of quicksort is $\Theta(n \log n)$, just like the worst-case running time of mergesort.
And the fact that quicksort tends to run faster than mergesort in practice,
probably indicates that the inputs that cause quicksort to take quadratic time
occur only rarely.

This illustrates how the average-case running time can be more realistic
than the worst-case running time. However, as we said earlier, the average-case
running time can be more difficult to determine because it requires knowledge
of the probability distribution of the inputs. In addition, average-case analysis
usually requires additional calculations. This was the case with the sequential
search algorithm, although the calculations there were still easy. The average-
case analysis of quicksort, on the other hand, is significantly more complicated
that its worst-case analysis.\footnote{We will do the worst-case analysis of quicksort later in these notes. The average-case
analysis is usually done in a course such as CS344 \textit{Algorithms and Data Structures}.}

In the rest of these notes, we will usually focus on the worst-case running
time of algorithms.

One final comment. In cases where even the worst-case analysis of an algo-
rithm proves difficult, it is possible to get an estimate of its asymptotic running
time by testing the algorithm on randomly generated inputs of various sizes
and seeing what kind of function best fits the data. But note that this gives
an estimate of the average-case running time, since there is no guarantee that
randomly generated inputs will include the worst-case ones. This kind of “em-
pirical analysis” can be especially useful if the average-case analysis is difficult
and we suspect that the worst-case running time may be too pessimistic.
Study Questions

8.7.1. What are the best-case, worst-case and average-case running times of an algorithm?

8.7.2. What is an advantage and a disadvantage of the worst-case running time compared to the average-case running time?

Exercises

8.7.3. Consider the class of vectors we implemented earlier in these notes. Determine the average-case running times of the various operations. For each operation, clearly identify what the input size is. For example, the input size of the assignment operator is the total size of the receiver and argument. In addition, state your assumptions about the distribution of inputs.

8.8 The Binary Search Algorithm

It is fairly obvious that searching a collection of data for a particular element, or for an element that satisfies a particular property, is a frequent operation. In this section, we will learn that under certain conditions, it is possible to search very efficiently by using an algorithm called binary search. We will also analyze the running time of this algorithm.

The simplest way of searching a sequence such as an array or a vector is to scan it from one end to the other, examining elements one by one. This is the sequential search we analyzed in the previous section. We found that its running time is linear in the length of the sequence.

If the sequence happens to be ordered, then the search can be done more quickly. For example, consider an array of integers sorted in increasing order. When looking for a particular integer, we can stop searching as soon as we find the integer we are looking for or an integer that is larger that the integer we are looking for. The running time of this modified sequential search is still linear but we can expect unsuccessful searches to be 50% faster, on average.

A much more dramatic improvement in the running time can be obtained for sorted sequences that provide constant-time access to their elements, such as arrays and vectors. The idea is to go straight to the middle of the sequence and compare the element we are looking for with the middle element of the
8.8. THE BINARY SEARCH ALGORITHM

Input: a sorted sequence s, an element e

while ( s contains more than one element ) {
    locate middle of s
    if ( e < middle element of s )
        s = left half of s
    else
        s = right half of s
}
compare e to only element in s

Figure 8.8: The binary search algorithm

s = [12 16 25 37 38 42 60 73] e = 25 middle = 38
[12 16 25 37]
[25 37]
[25]

Found!

Figure 8.9: A run of the binary search algorithm

sequence. Because the sequence is sorted, this comparison tells us if the el-
ment we are looking for should be located in the first or second half of the
sequence. We then only need to search that half.

This searching algorithm is called a **binary search**. The algorithm is
described in Figure 8.8.

Figure 8.9 shows a sample run of the algorithm on a sequence of integers.
The middle element is taken to be the one at the middle or to the immediate
right of the middle.

Figure 8.10 shows a generic implementation of the binary search algorithm
for arrays.

We now analyze the running time of the binary search algorithm under the
following two assumptions:

1. The middle element of the sequence can be accessed in constant time.

2. Elements can be compared in constant time.

For example, these assumptions are satisfied in the case of arrays and vectors
template <class T>
int binarySearch( const T a[], int start, int stop, 
    const T & e )
// Performs a binary search in a for e. Returns the index of 
// e in the range [start,stop). Returns -1 if value is not 
// found that range.
//
// PRECONDITION: The indices are valid and the elements in the 
// range [start,stop) are sorted in increasing order.
//
// ASSUMPTION ON TEMPLATE ARGUMENT: Values of type T can be 
// compared by using the < operator.
{
    int middle;
    int n = stop - start; // number of elements

    while ( n >= 2 ) {
        middle = ( start + stop ) / 2;

        if ( e < a[middle] ) stop = middle;
        else start = middle;

        n = stop - start;
    }

    if ( n == 1 ) {
        if ( e == a[start] ) return start;
        else return -1;
    }

    else return -1; // n <= 0
}

Figure 8.10: An implementation of binary search for arrays
that contain either integers or small strings.

Let $T(n)$ be the running time of the binary search algorithm on a sorted sequence of size $n$. To simplify the calculations, we first analyze the case when $n$ is a power of 2.

Consider the pseudocode shown in Figure 8.8. Since the step that follows the loop runs in constant time, the running time of the algorithm will clearly be dominated by the loop. Each iteration of the loop runs in constant time. Therefore, $T(n) = \Theta(r)$ where $r$ is the number of iterations of the loop.

We now determine what $r$ is. At every iteration of the loop, the size of the sequence decreases by half. Since we are assuming that $n$ is a power of 2, suppose that $n = 2^k$. After $i$ iterations, the size of the sequence is $n/2^i = 2^{k-i}$. The loop stops when the size reaches 1. Therefore, $r$ must satisfy $2^{k-r} = 1$, which implies that $r = k$. Of course, $k = \log n$, so that $T(n) = \Theta(\log n)$.

If $n$ is not a power of 2, then it falls between two consecutive powers of 2: $2^{k-1} < n < 2^k$. To show that $T(n)$ is still $\Theta(\log n)$, we can simply show that $T(n)$ is closely related to $T(2^{k-1})$ and $T(2^k)$. Clearly, the binary search algorithm takes more time on larger sequences. Therefore,

$$T(2^{k-1}) < T(n) < T(2^k)$$

The above analysis showed that $T(2^k) = ak + b$ where $a$ and $b$ are constants. Therefore,

$$a(k - 1) + b < T(n) < ak + b$$

All we have left to do is determine how $k$ relates to $\log n$. Since $2^{k-1} < n < 2^k$, we have that $k - 1 < \log n < k$. Therefore,

$$a(\log n - 1) + b < T(n) < a(\log n + 1) + b$$

because $k < \log n + 1$ and $k - 1 > \log n - 1$. Therefore, $T(n) = \Theta(\log n)$.

Note that our analysis of the binary search algorithm relies critically on the fact that the middle element of the sequence can be accessed in constant time. The binary search algorithm can also be used on other sorted sequences but, in that case, the running time may not be logarithmic. (An exercise asks you to explore this issue.)

Source code for the implementation of the binary search algorithm shown in this section is available on the course web site under binarySearch.cpp.
Exercises

8.8.1. Run the binary search sort algorithm on an array containing the following elements:

11  27  28  30  36  42  58  65

Search for elements 42 and 30. Illustrate each run of the algorithm as was done in Figure 8.9.

8.8.2. Suppose that computing the location of the middle element of a sequence takes time linear in the number of elements in the range currently being searched. (This is the case with linked lists, a data structure we will study later in these notes.) Show that the running time of the binary search algorithm is linear in this case.

8.8.3. Suppose that an array contains multiple copies of an element being searched for. As described in this section, the binary search algorithm will find the last occurrence of that element. Modify the algorithm so it finds the first occurrence. Verify your work by revising the implementation of the algorithm and testing it.
Chapter 9

Linked Lists

A linked list holds a sequence of elements all of the same type, just like an array or a vector. But linked lists can grow and shrink more efficiently. This makes them the data structure of choice for certain applications. In this chapter, we will learn what linked lists are and how to use them. The implementation of linked lists will be covered in a later chapter.

9.1 A Simple Text Editor

We will illustrate the usefulness of linked lists by designing and implementing a simple text editor. This editor will allow the user to add, remove and replace entire lines of text. It won’t allow the user to edit the contents of the lines. For example, it won’t be possible to insert a word directly in the middle of a line. The only way to accomplish this is will be by replacing the entire line by a new one.

Here are more details on how the editor works. The editor has a buffer that contains lines of text, usually an edited copy of the contents of some file. The editor displays a small number of lines from the buffer, no more than 10, for example, surrounded by some kind of border. A cursor indicates the position of the current line. Below the text, a menu of commands is displayed. Below that menu, a prompt such as “choice:”. The user types the first letter of a command, the command executes and everything is redisplayed. Some commands prompt the user for more information. Figure 9.1 shows what the user interface could look like.

The available editor commands are described in Figure 9.2. Note that the
displayed contents of the buffer always includes an extra empty line we will call the end line. That line is not really part of the buffer but the cursor can move there. This is how the user would insert a new line at the end of the buffer.

We will specify, design and implement the text editor later in this chapter. In this section, we tackle the main design issue: what data structure should the program use to store the contents of the buffer?

The buffer is really a sequence of lines. The lines are strings. So an array of strings might be our first idea. But an ordinary array is not a great choice because we don’t want to impose a limit on the number of lines in the buffer. And if we used a very large array, then the editor would use, perhaps most of the time, an unnecessarily large amount of memory.

This problem can be addressed, of course, by using a vector instead of an ordinary array. But is this really a good choice? An important issue is the impact that the use of a vector has on the running time of the various editor commands.

Some of the editor commands can be implemented quickly. For example, replace only requires replacing the current line in the vector by a new one.
9.1. A SIMPLE TEXT EDITOR

<table>
<thead>
<tr>
<th>Command</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>next</td>
<td>The next line becomes the current line.</td>
</tr>
<tr>
<td>previous</td>
<td>The previous line becomes the current line.</td>
</tr>
<tr>
<td>jump</td>
<td>Asks for a line number and makes that line become the current line.</td>
</tr>
<tr>
<td>replace</td>
<td>Asks for a new line and replaces the current line.</td>
</tr>
<tr>
<td>insert</td>
<td>Asks for a new line and inserts it before the current line.</td>
</tr>
<tr>
<td>delete</td>
<td>Deletes the current line.</td>
</tr>
<tr>
<td>open</td>
<td>Asks for a file name and reads that file into the buffer.</td>
</tr>
<tr>
<td>save</td>
<td>Asks for a file name and saves the contents of the buffer to that file.</td>
</tr>
<tr>
<td>quit</td>
<td>Stops the editor.</td>
</tr>
</tbody>
</table>

Figure 9.2: The commands of the text editor

Assuming that we store the index of the current line in an integer variable, replacing the current line can be done in constant time. The `next` command can also be executed in constant time, by simply increasing by one the index of the current line.

Other commands require much more work. The `insert` command, for example, requires that we first make room for the new line: before inserting a new line in the middle of the vector, we must first copy all the lines in the second half of the vector one position to the right. In the worst-case, this will take linear time.\footnote{We will learn later in these notes that there is a vector operation that can do that work for us. But the worst-case running time of that operation is still linear.} If the number of lines is large, we can expect that a linear-time command such as `insert` will be much slower than a constant-time command such as `replace` or `next`.

The second column of Table 9.1 shows the running times of all the editor commands if the buffer lines are stored in a vector. In this table, \( n \) stands for the number of lines in the buffer (plus those in the file, in the case of `open`).

The running time of the fast commands such as `replace` cannot be improved significantly. The linear running time of `open` and `save` is probably
something we have to live with since these commands require reading or writing every single line of the document. But do insert and delete really need to take linear time?

With a vector, these commands require a potentially time-consuming shifting of elements because vector elements are always stored side-by-side in the computer’s memory. This is what allows each element to be accessed quickly given its index.

An idea for a new data structure would therefore be not to store the lines of text side-by-side in the computer’s memory. Instead, we could try to scatter these lines in the computer’s memory and then somehow link them to each other. These links would be sequential in the sense that each line would be linked to the next one and to the previous one. We would maintain a link to the current line. Given this link, inserting a new line could be done quickly since it would only require modifying the links of the current and previous lines.

The idea we just described is that of a data structure known as a linked list. Later in this chapter, we will learn how to implement this idea by using pointers. We will also learn to use the STL container list, which is a class

<table>
<thead>
<tr>
<th></th>
<th>vector</th>
<th>linked list</th>
</tr>
</thead>
<tbody>
<tr>
<td>next</td>
<td>$\Theta(1)$</td>
<td>$\Theta(1)$</td>
</tr>
<tr>
<td>previous</td>
<td>$\Theta(1)$</td>
<td>$\Theta(1)$</td>
</tr>
<tr>
<td>jump</td>
<td>$\Theta(1)$</td>
<td>$\Theta(n)$</td>
</tr>
<tr>
<td>replace</td>
<td>$\Theta(1)$</td>
<td>$\Theta(1)$</td>
</tr>
<tr>
<td>insert</td>
<td>$\Theta(n)$</td>
<td>$\Theta(1)$</td>
</tr>
<tr>
<td>delete</td>
<td>$\Theta(n)$</td>
<td>$\Theta(1)$</td>
</tr>
<tr>
<td>open</td>
<td>$\Theta(n)$</td>
<td>$\Theta(n)$</td>
</tr>
<tr>
<td>save</td>
<td>$\Theta(n)$</td>
<td>$\Theta(n)$</td>
</tr>
<tr>
<td>quit</td>
<td>$\Theta(1)$</td>
<td>$\Theta(1)$</td>
</tr>
</tbody>
</table>

Table 9.1: Running time of the editor commands
of linked lists.

The main advantage of linked lists is that inserting or deleting an element at a given position can be done quickly. However, accessing an element given its index requires traversing the list from the beginning, counting elements until the desired one is reached. The third column of Table 9.1 shows the running times that can be achieved in our text editor if we use a linked list to store the lines of text in the buffer.

We clearly have a trade-off: we can have fast jumps but slow insertions and deletions, or we can have fast insertions and deletions but slow jumps. In the case of a text editor, it is reasonable to assume that insertions and deletions are more frequent than jumps, and that users are willing to accept that jumps require some “travel time”. Therefore, we choose a linked list for the text editor.

Study Questions

9.1.1. Why would an ordinary array be a poor choice for a data structure to store the lines of text in the editor?

9.1.2. What is the key difference between vectors and linked lists?

9.2 Linked Lists in the STL

The STL includes a class of linked lists that is simply called list. Just like vectors, STL lists are generic: even though each list holds a sequence of elements all of the same type, this type can be pretty much anything so that we can create lists of integers, lists of strings, lists of times, etc.

Tables 9.2 and 9.3 show some of the most basic list operations. The running time of these operations is also indicated. Unless otherwise specified, the parameter $n$ is the initial number of elements held by the receiver.

Included in these tables are several operations that allow the addition and removal of elements at the front and back of a list. But what about elsewhere in a list? For that purpose, we need a way of specifying a position within a list.

One way would be through indices, just like in arrays and vectors. But indexing is not an efficient operation with linked lists, as mentioned in the previous section. For example, an erase operation that took an index as
**Table 9.2: Some basic list operations**

<table>
<thead>
<tr>
<th>Operation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>list&lt;T&gt; ls</code></td>
<td>Creates a list <code>ls</code> that can hold elements of type <code>T</code>. The list is initialized to be empty, or to contain <code>n</code> copies of the default object of class <code>T</code>, or <code>n</code> copies of element <code>e</code>, or to be a copy of list <code>ls2</code>. $\Theta(1)$ for the first constructor, $\Theta(n)$ for the others. In the case of the last constructor, <code>n</code> is the size of <code>ls2</code>.</td>
</tr>
<tr>
<td><code>ls.size()</code></td>
<td>Asks list <code>ls</code> for the number of elements it currently contains. $\Theta(1)$.</td>
</tr>
<tr>
<td><code>ls.empty()</code></td>
<td>Asks list <code>ls</code> if it is empty. $\Theta(1)$.</td>
</tr>
<tr>
<td><code>ls.front()</code></td>
<td>Asks list <code>ls</code> for a reference to its front or back element. $\Theta(1)$.</td>
</tr>
<tr>
<td><code>ls.back()</code></td>
<td>Asks list <code>ls</code> for a reference to its front or back element. $\Theta(1)$.</td>
</tr>
<tr>
<td><code>ls.push_front(e)</code></td>
<td>Asks list <code>ls</code> to add a copy of element <code>e</code> to its front or back. $\Theta(1)$.</td>
</tr>
<tr>
<td><code>ls.push_back(e)</code></td>
<td>Asks list <code>ls</code> to add a copy of element <code>e</code> to its front or back. $\Theta(1)$.</td>
</tr>
<tr>
<td><code>ls.pop_front()</code></td>
<td>Asks list <code>ls</code> to delete its first or last element. $\Theta(1)$.</td>
</tr>
<tr>
<td><code>ls.pop_back()</code></td>
<td>Asks list <code>ls</code> to delete its first or last element. $\Theta(1)$.</td>
</tr>
<tr>
<td><code>ls.remove(e)</code></td>
<td>Asks list <code>ls</code> to delete all occurrences of element <code>e</code>. $\Theta(n)$.</td>
</tr>
<tr>
<td><code>ls.clear()</code></td>
<td>Asks list <code>ls</code> to delete all its elements. $\Theta(n)$.</td>
</tr>
<tr>
<td><code>ls1 = ls2</code></td>
<td>Makes list <code>ls1</code> a copy of list <code>ls2</code>. Returns a reference to <code>ls1</code>. $\Theta(n)$, where <code>n</code> is the total size of <code>ls1</code> and <code>ls2</code>.</td>
</tr>
</tbody>
</table>
9.2. LINKED LISTS IN THE STL

ls.resize(n)
ls.resize(n, e)
Asks list ls to change its size to n. If n is smaller than the current size of ls, the last elements of ls are deleted. If n is larger than the current size, then ls is padded with either copies of the default object of class T or with copies of element e. Θ(k), where k is the number of elements deleted or inserted.

ls.assign(n, e)
Asks list ls to change its contents to n copies of element e. Θ(m), where m is the greater of n and the initial size of ls.

ls1.swap(ls2)
Asks list ls1 to swap contents with list ls2. Θ(1).

Table 9.3: Some additional list operations

argument, as in ls.erase(i), would be required to step through all the elements of the list to find the one with index i.

What we need is a mechanism that not only marks a position within a list but also gives us efficient access to the element at that position. It would also be useful if that mechanism allowed us to efficiently move back and forth through the elements of the list.

Such a mechanism is provided in the form of iterators. An iterator is a value (usually an object) that points to an element in a container, gives access to that element, and can be moved to point to the next or previous element. Thus iterators can be used to iterate through all the elements of a container, from beginning to end.

Figure 9.3 shows a generic code fragment that prints the contents of a list of elements of type T. Let’s examine this code carefully.

The loop starts by declaring an iterator itr. Lists provide two basic iterator types: iterator and const_iterator. The only difference is that a constant iterator cannot be used to change the element it points to. Iterator types defined by lists can be accessed by using the double-colon syntax (::), as shown in Figure 9.3. Constant iterators are used here simply for safety reasons, to prevent the loop from accidentally modifying the list.

The iterator is then initialized to a begin iterator, which is an iterator that
for ( list<int>::const_iterator itr = ls.begin();
    itr != ls.end();
    ++itr ) {
    cout << *itr << endl;
}

Figure 9.3: Printing the contents of a list

points to the first element of the list. Note how the begin iterator is obtained from the list by using the begin method.

In the body of the loop, the dereferencing operator (*) is used to access the list elements. We said earlier in these notes that if p is a pointer, then *p refers to the value that p points to. The same is true with iterators: if itr is an iterator, then *itr refers the value that the itr points to. In this case, *itr is a list element.

Next, the iterator is moved to point to the next list element by using the increment operator ++. Note that if needed, an iterator can be moved back to the previous element by using the decrement operator --.

The loop stops as soon as itr becomes equal to the end iterator. In general, in the STL, an end iterator does not point to the last element of a container. In fact, it does not point to any element at all. The end iterator is just a marker that indicates that the end of the container has been reached. The end iterator can be viewed as pointing to a position that’s just past the last element.

Table 9.4 shows some list operations that involve iterators. Two of these operations refer to a range [start, stop). In the STL, a range of positions or elements is specified by two iterators that act as the end points of an interval. The first iterator is included in the range but the second one is not, as indicated by the interval notation [start, stop).

Table 9.5 summarizes the iterator operations described in this section. More will be said about iterators later in these notes.

The list container is defined in library file list and included in the std namespace. STL maps include several additional operations that are described in a reference such as cplusplus.com [CPP].
9.2. LINKED LISTS IN THE STL

`ls.begin()`  
Asks list `ls` for a begin or end iterator. $\Theta(1)$.

`ls.end()`  
Asks list `ls` for a begin or end iterator. $\Theta(1)$.

`ls.insert(itr, e)`  
Asks list `ls` to insert, at the position indicated by the iterator `itr`, a copy of element `e`. An iterator that points to the new element is returned. $\Theta(1)$.

`ls.insert(itr, start, stop)`  
Asks list `ls` to insert, at the position indicated by the iterator `itr`, copies of all the elements in the range `[start, stop)`. $\Theta(k)$, where $k$ is the number of elements inserted.

`ls.erase(itr)`  
Asks list `ls` to delete the element that `itr` points to. An iterator that points to the next element is returned. $\Theta(1)$.

`ls.erase(start, stop)`  
Asks list `ls` to delete all the elements in the range `[start, stop)`. $\Theta(k)$, where $k$ is the number of elements deleted.

<table>
<thead>
<tr>
<th>Table 9.4: Some list operations that involve iterators</th>
</tr>
</thead>
<tbody>
<tr>
<td>*itr</td>
</tr>
<tr>
<td><code>++itr</code></td>
</tr>
<tr>
<td><code>--itr</code></td>
</tr>
<tr>
<td><code>itr1 == itr2</code></td>
</tr>
<tr>
<td><code>itr1 != itr2</code></td>
</tr>
<tr>
<td>Returns <code>true</code> if <code>itr1</code> and <code>itr2</code> point or don’t point to the same element.</td>
</tr>
</tbody>
</table>
CHAPTER 9. LINKED LISTS

Study Questions

9.2.1. What mechanism is used to specify positions within an STL list?

9.2.2. What is an iterator?

9.2.3. What is a constant iterator?

9.2.4. Does it make sense to dereference the end iterator of a container?

9.2.5. Do the following two statements have the same effect: `ls.clear()` and `ls.erase( ls.begin(), ls.end() )`?

Exercises

9.2.6. Experiment with lists by writing a test driver that creates more than one type of list and uses all the methods shown in Tables 9.2 to 9.4.

9.2.7. Write a generic function

```cpp
int count( const list<T> & ls, T e )
```

that returns the number of occurrences of element `e` in list `ls`. *Hint:* Because `ls` is constant, you will need to use constant iterators.

9.2.8. Write a generic function

```cpp
list<T>::iterator find( list<T> & ls, T e )
```

that returns an iterator to the first occurrence of element `e` in list `ls`. In case `e` is not found, the end iterator is returned.

9.2.9. Write a function

```cpp
list<string>::iterator find_substring(
    list<string> & ls, const string & s )
```

that returns an iterator to the first string in list `ls` that contains string `s` as a substring. In case `s` is not found, the end iterator is returned. Use the string `find` method. (See Section A.2.)

9.2.10. Write a generic function
9.3 Design and Implementation of the Text Editor

In the first section of this chapter, we decided that a linked list would be better than an array for storing the lines of text in our editor. In the previous section, we learned how to use the class of linked lists available in the STL. In this section, we take a quick look at how to use STL lists in the implementation of the text editor.

The general behavior of the text editor was described in Section 9.1. But several details still need to be specified. For example, what should the next command do if the current line is the end line? What line should be the current one after an insert? What should happen if a file doesn’t open? A full specification for a first version of the editor is available on the course web site under TextEditor1.0. It includes answers to these questions as well as several other details.

Also available there is a complete design document. There are three main tasks that the editor needs to handle:

- the user interaction
- the storage of the buffer
- the execution of the commands

Ideally, we would like to assign these tasks to three separate components. But the execution of the commands is highly dependent on the exact way in which the buffer is stored, so it makes sense to assign these two tasks to a single component that we call Buffer. The overall control of the editor, and most of the user interaction, is assigned to a TextEditor component.

The TextEditor has a single public method: run. The TextEditor stores the lines of text in a Buffer object. The displaying of the buffer and the execution of the commands is delegated to the Buffer object.
class TextEditor
{
public:
    void run();

private:
    void display() const;

    Buffer buffer;
};

Figure 9.4: Declaration of TextEditor

The Buffer object has several public methods, one for displaying the buffer and one for each of the editor commands. The Buffer object stores the lines of text in a linked list, as decided in Section 9.1.

We now examine some aspects of the implementation of the program. Figure 9.4 shows the declaration of the TextEditor class. The private display method is a helper method to which run delegates the displaying of the buffer and of the menu of commands. Figure 9.5 shows a portion of the run method. The code is fairly straightforward.

Figure 9.6 shows the declaration of the Buffer class. The iterator it_current_line marks the location of the current line. It is initialized to the end iterator, which we view as pointing to the end line. The class declaration also includes the implementation of the two move methods as well as the insert and erase methods. Figure 9.7 show the implementation of the open method.

The complete source code and documentation for this version of the text editor is available on the course web site under TextEditor1.0.

Exercises

9.3.1. Modify the text editor as described below. Change the original specification, design and implementation as little as possible.

   a) Implement the replace command.

   b) Implement the jump command. Implement this as efficiently as possible. *Hint:* Store the number of the current line. You’ll need
void TextEditor::run()
{
    bool done = false;
    do {
        display();

        cout << "choice: ";
        char command;
        cin >> command;
        cin.get(); // new line char

        switch (command) {
        case 'i': {
            cout << "new line: ";
            string new_line;
            getline(cin, new_line);
            buffer.insert(new_line);
            break;
        }

        case 'n': {
            buffer.move_to_next_line();
            break;
        }

        case 'q': {
            done = true;
            break;
        }
        }
    } while (!done);

    cout << endl;
    return;
}

Figure 9.5: A portion of run
class Buffer
{
public:
    Buffer() { it_current_line = ls_lines.end(); }

    void display() const;

    void move_to_next_line() {
        if ( it_current_line != ls_lines.end() )
            ++it_current_line;
    }

    void move_to_previous_line() {
        if ( it_current_line != ls_lines.begin() )
            --it_current_line;
    }

    void insert( const std::string & new_line ) {
        ls_lines.insert( it_current_line, new_line );
    }

    void erase() {
        if ( it_current_line != ls_lines.end() )
            it_current_line = ls_lines.erase( it_current_line );
    }

    bool open( const std::string & file_name );
    bool save( const std::string & file_name ) const;

private:
    std::list<std::string> ls_lines;
    std::list<std::string>::iterator it_current_line;
};

Figure 9.6: Declaration of Buffer
bool Buffer::open( const string & file_name )
{
    ifstream file( file_name.c_str() );
    if ( !file ) return false;

    ls_lines.clear();
    // Note: the list is cleared only after we know the file
    // opened successfully.

    string line;
    while ( getline(file, line) ) ls_lines.push_back(line);
    file.close();
    it_current_line = ls_lines.begin();
    return true;
}

Figure 9.7: Implementation of open

to revise the other methods accordingly.

c) The text editor currently always displays the entire buffer. Modify
the program to display only 9 lines with the current line as close as
possible to the center.

d) The quit command currently stops the editor without ensuring
that the buffer was saved to a file. Similarly, the open command
reads new contents into the buffer without ensuring that the current
contents of the buffer was saved. Fix this. In case the buffer has not
been saved since it was last modified, open and quit should ask
the user if he or she wants to save. Hint: Add a Boolean variable
to the buffer that indicates if the buffer was saved since the last
modification.

e) Add an append command that inserts a new line after the current
one. Make sure you consider special cases (such as those for insert
and delete).

f) Add a search command that finds the first line that contains some
string entered by the user. The search starts at the current line and
stops at the end of the document. If the string is found, the line
that contains it becomes the current line. If the string is not found,
a message is printed and the current line stays where it is. Use the string find method. (See Section A.2.)
Chapter 10

Maps

In this chapter, we will learn to use maps, another container included in the STL. As a sample application, we will use maps to create a phone book program.

10.1 A Phone Book

To illustrate the usefulness of maps, we will create a simple phone book program. Entries in this phone book will consist of only a name and a phone number. The user will be able to browse the phone book, search for an entry, as well as add, edit and delete entries.

Figure 10.1 shows what the user interface looks like. The program displays a single entry. Below the entry, a menu of commands is displayed. Below that menu, the prompt “command:” is displayed. The user types the first letter

John Smith
123–456–7890

next search edit quit

previous add delete

command: e
new number:

Figure 10.1: Sample user interface of the phone book program
Some commands prompt the user for more information. The available phone book commands are described in Figure 10.2. The entries are displayed in alphabetical order. Note that no entry is displayed if the phone book is empty.

We will design and implement the phone book later in this chapter. In this section, we consider one of the main design issues: how should the program store the entries of the phone book? Between runs of the program, the entries will need to be stored in a file. But while the program is running, it is much more efficient to copy the entries into main memory (that is, the variables of the program).

The simplest idea is to store the entries in alphabetical order in a vector. And the most convenient way of doing that is to define a class of phone book entries. Each object in this class will hold the name and number of one entry. We can then store all the entries in a single vector of entry objects.

In addition, if we make sure the program keeps the phone book entries sorted by name, then we can search for entries by using the binary search algorithm. If $n$ is the number of entries in the phone book, searches would run in time $\Theta(\log n)$, which is much better than the $\Theta(n)$ time we would get with
10.1. A PHONE BOOK

<table>
<thead>
<tr>
<th>Data Structure</th>
<th>Search</th>
<th>Add</th>
<th>Delete</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sorted vector plus binary search</td>
<td>$\Theta(\log n)$</td>
<td>$\Theta(n)$</td>
<td>$\Theta(n)$</td>
</tr>
<tr>
<td>Balanced binary search tree</td>
<td>$\Theta(\log n)$</td>
<td>$\Theta(\log n)$</td>
<td>$\Theta(\log n)$</td>
</tr>
<tr>
<td>Hash table</td>
<td>$\Theta(1)$</td>
<td>$\Theta(1)$</td>
<td>$\Theta(1)$</td>
</tr>
</tbody>
</table>

Table 10.1: Summary of the various options discussed in this section

...a sequential search.

But adding and removing entries from the phone book would require linear time. For example, when adding an entry, all existing entries to the right of the new entry would need to be shifted one position to the right. And there seems to be no way around this.

There are data structures that give us fast searches, additions and deletions. One of them is the balanced binary search tree. This is actually a general category of data structures that includes, for example, red-black trees and AVL trees. Balanced binary search trees support all three operations in time $\Theta(\log n)$.

Another option is the hash table. Hash tables can be implemented in different ways but a simple implementation known as separate chaining supports searches and deletions in constant time on average. Additions can be performed in amortized constant time. This means that starting with an empty hash table, a sequence of $n$ additions will take time that averages to a constant per addition, if $n$ is large enough. In other words, a sequence of $n$ additions will take time $\Theta(n)$ in total.

At first glance, hash tables look superior to balanced binary search trees. However, hash tables require fine tuning that is often application-dependent. In addition, the worst case running time of the operations is $\Theta(n)$. Even though it is unlikely, it is always possible that a large number of operations will end up taking that much time. Balanced binary search trees, on the other hand, require no fine tuning and have guaranteed logarithmic time operations.

Table 10.1 summarizes the three options we have discussed in this section. The implementation details of the last two data structures are beyond the scope of these notes.1 But the STL includes a container called `map` that

---

1At Clarkson, these data structures are covered in detail in the course CS344 Algorithms
guarantees the performance of balanced binary search trees and is typically
implemented using that data structure.\footnote{The STL does not currently include a hash table container but one may be included in an upcoming revision of the STL specification.}

If the number of entries is not too large, the phone book program would probably perform well under any of these three options. If the number of entries was very large, balanced binary search trees or hash tables would be preferable. In C++, the most convenient option is to use an STL map. This is also more reliable than trying to implement our own hash tables from scratch.

In this chapter, we will learn how to use STL maps. We will then create a version of the phone book program that uses maps.

In fact, there is little reason to use a sorted vector instead of a map. One reason would be if memory space was very tight. This is because even though a balanced binary search tree with $n$ elements uses $\Theta(n)$ memory, it uses more memory than the optimal amount of a sorted vector of just the right size.

Another reason for using a vector instead of a map would be if the elements were indexed by consecutive integers. In the phone book program, entries are indexed by name, that is, the name of an entry is what is used to access it. But imagine a program that runs a time-intensive algorithm on a graph (such as a network) and needs quick access to information about the nodes in that graph. If the nodes are numbered 1 through $n$, then the nodes can be stored in a vector and accessed in constant time. With a map, access would take logarithmic time. If the number of nodes is large, the difference can be significant.

However, these situations are not that common. Typically, for indexed data, maps are the data structure of choice.

\section{Maps in the STL}

As mentioned in the previous section, the STL includes a container called map that guarantees the performance of balanced binary search trees and is typically implemented by using that data structure. A map stores a collection of elements that each have a key. Map elements are indexed using those keys, which means that the elements are normally accessed by using those keys. For example, in the phone book program, the names would be used as keys to access the phone book entries.
10.2. MAPS IN THE STL

```
for ( int i = 0; i < n; ++i ) {
    cout << a[i] << endl;
}
```

Figure 10.3: Printing the contents of an array

Keys are similar to array or vector indices and in that sense, maps are somewhat similar to arrays and vectors. But a critical difference is that map keys do not have to be integers.

The name `map` comes from the fact that each map associates a unique element to each key, just as a mathematical map or function would. Maps are sometimes also called `dictionaries` (with the keys viewed as words and the elements as definitions) or `tables` (with two columns, one for the keys, the other for the elements).

In principle, map keys could be any type of value. But STL maps require that keys support the `<` operator. This makes it possible for STL maps to be implemented as binary search trees.

Maps are generic because both keys and elements can be of a variety of types. Therefore, when declaring a map, two types must be specified: one for the keys and another for the elements. For example, `map<string, int>` is a type of map in which keys are strings and elements are integers. In other words, maps of integers indexed by strings.

Map elements can be accessed by using the usual indexing operator. For example, `m[k]` returns a reference to the element whose key is `k`. The indexing operator can also be used to add elements to a map: if no element with the given key is present in the map, then the indexing operator will automatically create one. That element will be the default object of class `T`, or a random value in case `T` is not a class. Usually, this default element is immediately replaced by a copy of some other element as in `m[key] = e`.

Now that we know how to add and access map elements, one key question is how do we print the contents of a map? With an array of size `n`, we know that the indices range from 0 to `n - 1`. This allows us to write a simple loop that accesses all the elements by using indices and the indexing operator, as shown in Figure 10.3. But map keys are not necessarily consecutive values over some range.

So we need a mechanism that allows us to step through all the elements of a map. And we have just seen such a mechanism in the previous chapter:
for ( map<K, T>::const_iterator itr = m.begin();
    itr != m.end();
    ++itr ) {
    cout << itr->first << " : " << itr->second << endl;
}

Figure 10.4: Printing the contents of a map

iterators. Recall that an iterator is a value (usually an object) that points to an element in a container, gives access to that element, and can be moved to point to the next or previous element. Therefore, as we saw in the case of lists, iterators can be used to step through all the elements of a container. Figure 10.4 shows a code fragment that prints the contents of a map that contains elements of type T indexed by keys of type K.

A key difference between map and list iterators is that map iterators point to a pair key–element. Each map pair is an object of class pair<const K, T>. The class pair is a standard STL class. In general, each object of class pair<First, Second> combines two values, one of type First, the other of type Second. Each pair has two public data members first and second that hold the two components of the pair. In a map, first is the key and second is the element.

In the body of the loop in Figure 10.4, the dereference-and-select operator -> is used to access the keys and elements. This operator, which is often called the arrow operator, is a convenient combination of the dereferencing operator (*) followed immediately by the selection operator (.). In fact,

itr->first

could have been written

(*itr).first

It is important to note that when using iterators to traverse a map, the pairs are encountered in increasing order of key. This goes with the fact that maps are usually implemented as binary search trees. But it’s also convenient. In particular, it means that STL maps can be viewed (and used) as sequences of elements ordered by key.

Tables 10.2 and 10.3 show some of the most basic map operations. Their running times are indicated as functions of n, the number of elements in the
map. These running times assume that keys of type K and elements of type T
can be copied and compared in constant time.

The insert and second erase operations run in amortized constant time. Recall that this means that over the long run, the running time per operation
will average to a constant.

The map container is defined in library file map and included in the std
namespace. The pair class is defined in library file utility, which is always
included in map. STL maps include several additional operations that are
described in a reference such ascplusplus.com [CPP].

Study Questions

10.2.1. What is a map?

10.2.2. How many types need to be specified when creating a map? What do they represent?

10.2.3. What happens when the indexing operator is used with a key that is not present in a map?

10.2.4. What type of value do map iterators point to?

Exercises

10.2.5. Experiment with maps by writing a test driver that creates more than one type of map and uses all the methods shown in Table 10.2.

10.2.6. Suppose that a file contains the names and ages of various people. Each name is given on a line by itself. The following line contains the age of the person. Write code fragments that perform the following:

a) Read the file and store all the ages of the people in a map indexed by name.

b) Print the contents of the map with one line per person in the following format:

   Joe Smith: 45

c) Print the names of all the people who are younger than 21.
map<K, T> m
map<K, T> m(m2)

Creates a map m that can hold elements of type T with keys of type K. The map is initialized to be empty, or to be a copy of map m2. \( \Theta(1) \) for the default constructor. \( \Theta(n) \) for the copy constructor, where \( n \) is the size of m2.

m.size()

Asks map m for the number of elements it currently contains. \( \Theta(1) \).

m.empty()

Asks map m if it is empty. \( \Theta(1) \).

m.begin()

m.end()

Asks map m for a begin or end iterator. \( \Theta(1) \).

m[k]

Asks map m for a reference to the element with key k. If no such element exists, one is created. \( \Theta(\log n) \).

m.find(k)

Asks map m for an iterator to the pair with key k. If no such pair exists, the end iterator is returned. \( \Theta(\log n) \).

m.erase(k)

Asks map m to delete the element with key k, if one exists. Returns 1 if an element was deleted; 0, otherwise. \( \Theta(\log n) \).

m.erase(itr)

Asks map m to delete the pair that itr points to. Amortized \( \Theta(1) \).

m.erase(start, stop)

Asks map m to delete all elements in the range \([\text{start, stop}]\). \( \Theta(\log n + r) \), where \( r \) is the number of elements removed.

m.clear()

Asks map m to delete every element from map m. \( \Theta(n) \).

Table 10.2: Some map operations
10.2. MAPS IN THE STL

m.count(k)
Asks map m for the number of elements it contains that have key k. Either 0 or 1 is returned. $\Theta(\log n)$.

m.insert(itr, p)
Asks map m to insert a copy of pair p, which consists of a key and an element. No insertion occurs if the map already contains a pair with p’s key. Returns an iterator that points to the new pair. The iterator itr is an indication of where the new pair may belong. Amortized $\Theta(1)$ if the new pair is inserted right after the position indicated by itr. Otherwise, $\Theta(\log n)$.

m1 = m2
Makes map m1 a copy of map m2. Returns a reference to m1. $\Theta(n)$, where n is the total size of m1 and m2.

m1.swap(m2)
Asks map m1 to swap contents with map m2. $\Theta(1)$.

Table 10.3: Some additional map operations
10.2.7. Suppose that a file contains the numbers and prices of various products. Each product number is given on a line by itself. The following line contains the price of the product. Write code fragments that perform each of the following:

a) Read the file and store all the prices of the products in a map indexed by product number.

b) Print all the products in the map that cost less than $1.

c) Create another map that contains all the products that cost less than $1.

10.3 Design and Implementation of the Phone Book

The first section of this chapter described the behavior of the phone book program and addressed some of its design. We now finish the design and implement the program.

As decided earlier, we will have a class of phone book entries that will each hold a name and a phone number. These objects will be responsible for their own reading and printing (just like Time and string objects).

The design and implementation of this class are shown in Figures 10.5 and 10.6. Note that we are not hiding the data members of the class (by making them private). There is little advantage to hiding these data members since it is unlikely that the way in which these strings are stored will change.

The actual running of the phone book will be handled by a separate component of the program, a class called PhoneBook. The class will store the phone book entries in a map of PhoneBookEntry objects indexed by name. The class will have one public method called run and several private helper methods.

Note that it is not necessary to use a class for the PhoneBook component. That component could have been a collection of standalone functions consisting of the run function and its helpers. The map would have been a local variable of run. The main disadvantage of this approach is that every time run would call one of these helpers, it would need to pass to the helper the map of entries (and possibly other variables). In contrast, with a class, run and all its methods have direct access to the map; there is no need to pass it
COMPONENT: class PhoneBookEntry

Each object represents a phone book entry.

Data members (public):

    string name
    string number

Stand-alone functions:

    istream & operator>>( istream & in, PhoneBookEntry & e )
    Reads e from in. A phone book entry consists of two
    lines, one for the name, one for the number. Returns
    in. No error-checking.

    ostream & operator<<( ostream & out,
                     const PhoneBookEntry & e )
    Prints e to out in the format described for read.
    Returns out.

Collaborators: none.

Figure 10.5: The design of the class PhoneBookEntry
class PhoneBookEntry
{
    public:
        PhoneBookEntry() {
            name = "no name";
            number = "no number";
        }

        PhoneBookEntry( const std::string & name0, const std::string & number0 ) {
            name = name0;
            number = number0;
        }

        std::string name;
        std::string number;
};

inline std::istream & operator>>( std::istream & in, PhoneBookEntry & e ) {
    getline( in, e.name );
    getline( in, e.number );
    return in;
}

inline std::ostream & operator<<( std::ostream & out, const PhoneBookEntry & e ) {
    out << e.name << std::endl << e.number << std::endl;
    return out;
}

Figure 10.6: The declaration and implementation of PhoneBookEntry


COMPONENT: class PhoneBook

A simple phone book. See program spec for details.

Method:

    void run()
    Runs the phone book. See program spec for details.

Implementation: The run method reads the entries from a file and stores them in a map of PhoneBookEntry objects indexed by name. This map is a data member of the PhoneBook class. Then, in a loop, run displays the current entry, the menu of commands, reads a command and executes it. The file is called phonebook.txt. The first line of the file contains the number of entries in the phone book. The reading and printing of entries is delegated to the PhoneBookEntry objects.

Alternative: Phone book data could have been stored in a map of numbers indexed by name. One advantage: less memory because names are not stored twice. Two disadvantages: (1) harder to print stored entries; (2) harder to extend program to entries that hold more than just a name and phone number.

Collaborators: PhoneBookEntry.

Figure 10.7: The design of the PhoneBook class

around. Even though this approach introduces more dependencies, it makes sense here since all of these methods can honestly be viewed as sharing access to the map.

Figure 10.7 describes the design of the PhoneBook class. This include some notes on the implementation of the class. The helper methods could have been documented here but we decided instead to leave the discovery of those methods to the person implementing the class. The design of PhoneBook also documents an alternative way of storing the phone book entries and the reason behind our choice. This is likely to be useful later, after we have forgotten the rationale behind our design.
class PhoneBook
{
public:
    PhoneBook() : ksFileName("phonebook.txt") { init(); }

    void run();

private:
    std::map<std::string, PhoneBookEntry> m_entries;
    std::map<std::string, PhoneBookEntry>::iterator itr_current_entry;
    const std::string ksFileName;

    void init() { itr_current_entry = m_entries.end(); } // Initializes non-constant data members.

    void display_entry_and_menu() const;
    void execute(char command, bool & done);

    void read_file(); // Reads entries from file and stores them in arrays.
    // Allocates the arrays. Assumes no arrays to deallocate.

    void write_file() const;
};

Figure 10.8: The declaration of the PhoneBook class

Figure 10.8 shows the declaration of the PhoneBook class. The data members include an iterator to the copy of the current entry stored in the map. Several private helper methods are declared and documented.

The class declaration also includes the implementation of the default constructor. The file name is initialized by using what is called an initializer. This is necessary because the data member ksFileName is declared constant, which implies that it cannot be set by using the assignment operator. In situations where multiple initializers are needed, they should be separated by commas, as in

dataMember1(value1), dataMember2(value2)
void PhoneBook::run()
{
    read_file();
    bool done = false;
    do {
        display_entry_and_menu();
        cout << "choice: ";
        char command;
        cin >> command;
        cin.get();  // new line char
        execute( command, done );
        cout << endl;
    } while ( !done );
}

Figure 10.9: The run method

The body of the constructor consists of a call to the method init. Note how the receiver of init is implied. Again, the easiest way to make sense of this is to imagine that a method tells an object how to respond to a message: the constructor is telling its receiver to initialize itself.

Figure 10.9 shows the implementation of run. This code is relatively straightforward.

Figure 10.10 shows a portion of execute. This method is essentially a big switch statement. The next command causes the iterator itr_current_entry to move to the next entry, unless doing so would make it become the end iterator. In that case, the iterator is set to point to the first entry.

Note the use of the two different dereferencing operators (\-> and .) in the implementation of the edit command. This is necessary since itr_current_entry points to a pair while second refers to a phone book entry.

Figure 10.11 shows another portion of execute. The add command reads a new name and phone number from the user, creates a new phone book entry, combines the new name and new entry into a new pair, and inserts that pair into the map. Note that the insert method is convenient here since we need to set itr_current_entry to point to the new entry and the insert method returns precisely such an iterator.
void PhoneBook::execute( char command, bool & done )
{
    switch ( command ) {
    case 'n': {
        if ( m_entries.empty() ) return;
        ++itr_current_entry;
        if ( itr_current_entry == m_entries.end() ) {
            itr_current_entry = m_entries.begin();
        }
        break;
    }
    case 'e': {
        if ( m_entries.empty() ) return;
        cout << "new number: ";
        string new_number;
        getline( cin, new_number );
        itr_current_entry->second.number = new_number;
        break;
    }
    case 's': {
        cout << "name: ";
        string name;
        getline( cin, name );
        map<string, PhoneBookEntry>::iterator
        itr = m_entries.find( name );
        if ( itr != m_entries.end() )
            itr_current_entry = itr;
        break;
    }
    ...  
    };
}

Figure 10.10: The execute method (part 1 of 2)
```cpp
void PhoneBook::execute( char command, bool & done )
{
    switch ( command ) {
        ...
        case 'a': {
            cout << "new name: 
            string new_name;
            getline( cin, new_name );
            cout << "phone number: 
            string new_number;
            getline( cin, new_number );
            PhoneBookEntry new_entry( new_name, new_number );
            pair<string, PhoneBookEntry> new_pair( new_name, new_entry );
            m_entries.insert( m_entries.end(), new_pair );
            break;
        }
        case 'q': {
            write_file();
            m_entries.clear();
            init();
            done = true;
            break;
        }
    }
}
```

Figure 10.11: The execute method (part 2 of 2)
void PhoneBook::read_file() {
    ifstream ifs( ksFileName.c_str() );
    if ( !ifs ) return; // no file; one will be created on quit

    int num_entries;
    ifs >> num_entries;
    ifs.get(); // \n
    map<string, PhoneBookEntry>::iterator last = m_entries.end();
    for ( int i = 0; i < num_entries; i++ ) {
        PhoneBookEntry new_entry;
        ifs >> new_entry;
        m_entries[ new_entry.name ] = new_entry;
    }
    itr_current_entry = m_entries.begin();
    ifs.close();
}

Figure 10.12: A possible implementation of the read_file method

Figure 10.11 also shows the implementation of the quit command. After writing the entries to the file, the array is deallocated and the data members are reset to their initial values. This returns the phone book to its initial state and ensures that the PhoneBook object will behave correctly if it is asked to run again.

Figure 10.12 shows a possible implementation of the PhoneBook helper method read_file. After opening the file and reading the number of entries, entries are read and added to the map one by one by using the indexing operator.

This code is simple but not the most efficient. Suppose that \( n \) is the total number of entries in the file. Once the map is half full, each indexing operation will take time at least \( \Theta(\log(n/2)) = \Theta(\log n - 1) = \Theta(\log n) \). This leads to a total reading time that’s at least \( \Theta(n \log n) \). (It’s also clear that the reading time is no greater than \( \Theta(n \log n) \) since each indexing operation takes no more than \( \Theta(\log n) \) time.) In contrast, if entries were added to an unsorted vector, the reading could be done in time \( \Theta(n) \).

The running time of read_file can be improved by using the map insert operation. The key idea is to make sure that the entries in the file are always sorted in increasing order by name. This way, when we read the entries, we
void PhoneBook::read_file() {
    ifstream ifs( ksFileName.c_str() );
    if ( !ifs ) return; // no file; one will be created on quit

    int num_entries;
    ifs >> num_entries;
    ifs.get(); // 
    map<string, PhoneBookEntry>::iterator last = m_entries.end();
    for ( int i = 0; i < num_entries; i++ ) {
        PhoneBookEntry new_entry;
        ifs >> new_entry;
        pair<string, PhoneBookEntry>
        new_pair( new_entry.name, new_entry );
        last = m_entries.insert( last, new_pair );
    }
    itr_current_entry = m_entries.begin();
    ifs.close();
}

Figure 10.13: A more efficient implementation of read_file

know where each new entry needs to be inserted: at the end of the map. So if
each new pair is inserted as follows

    m_entries.insert( last, new_pair );

with last pointing to the last pair in the map, then each insertion will run in
amortized constant time, for a total reading time of $\Theta(n)$.

Figure 10.13 shows the implementation of read_file revised accordingly.
The initial value of last does not matter so we use the end iterator. After the
insertion of each pair, last is updated to the return value of insert, which
is an iterator that points to the newly inserted pair.

The source code and documentation of the phone book program are avail-
able on the course web site under PhoneBook1.0. Note that the implementa-
tion of the program is not complete; one of the exercises asks you to add
what’s missing.
Study Questions

10.3.1. What is the advantage of making the component that runs the phone book a class instead of a standalone function with helpers?

10.3.2. What is an initializer used for?

Exercises

10.3.3. Modify the search command of the phone book program as follows: in case an entry is not found, the program should display the entry that would normally follow the one that was searched for. For example, in a phone book with Alice and Charlie, searching for Bob would show Charlie. Hint: Use the lower_bound method. See [CPP] for details. Make sure to consider the case when the name searched for is larger than all the names currently in the phone book.

10.3.4. Complete the implementation of the phone book program by writing code for the following commands.

   a) previous.

   b) quit. (All that is left to do is to implement the helper method write_file.)

   c) delete.
Chapter 11

Object-Oriented Design

We have already created four programs in these notes: a pay calculator, a spell-checker, a text editor and a phone book. These programs allowed us to learn a number of important concepts and techniques. In particular, we learned about data abstraction, classes, dynamically allocated arrays, vectors, linked lists and maps. In this chapter, we will create a somewhat larger program, a cash register. This will allow us to discuss the software development process in more detail and will also be a good example of the usefulness of maps.

11.1 The Software Life Cycle

Figure 11.1 shows an initial description of the cash register program we will create in this chapter. This program will have to be specified, designed and implemented, as we saw when we created the pay calculator and the text editor earlier in these notes.

Software specification involves determining exactly what the software must do. A specification is normally concerned only with the behavior of the software as seen from the outside. In other words, a specification spells out what the software must do, not how it does it. As mentioned in Chapter 1, a good specification should be clear, correct and complete. It also helps if it is as concise as possible. Specifying software typically involves communicating with the client.

Software design generally consists of the following three tasks:

1. Identify the various components of the software and the tasks they are responsible for.
Typical session:

(n)ew costumer  (a)dd to previous order  (c)lose register
? n

Enter product number and quantity (0 = end)

> 4237 1
   Bananas          1.25

> 5031 3
   3 @ 1.19
   Chickpeas       3.57

> 0

4 items  Total:  4.82

(n)ew costumer  (a)dd to previous order  (c)lose register
? ...

Data file:

4237 1.25 Bananas
5031 1.19 Chickpeas
...

Figure 11.1: The initial description of the cash register program
2. Choose or design major algorithms and data structures for these components.

3. Write precise specifications for these components, including precise interfaces.

The interface of a component is what client code uses to communicate with it. In the case of a function, this consists of the name of the function as well as the type of its arguments and its return value. In the case of a class, this consists of the name of the class and the interface of all of its public methods and data members.

In other words, after the software is designed, we should know what components it contains, what these components do and how to use them.

Software implementation is the writing and testing of the code. This normally involves the following:

1. Code each software component.

2. Test each component on its own. As we saw in Chapter 1, this is called *unit testing*.

3. Combine the components gradually, one at a time, testing after each addition. This is called *integration testing*.

Unit and integration testing make it easier to locate and fix errors. This is especially important when dealing with large programs.

Once a program is implemented, it is ready to be used. But the software will usually continue to evolve. This can consist of fixing errors (as reported by users, for example), adapting the software to new environments (such as new hardware), extending the software (to give it new capabilities), or more generally improving the software (to make it more efficient or easier to use, for example). All of these activities constitute what is called *software maintenance and evolution*, or just software maintenance, for short.

Software specification, design, implementation and maintenance are often referred to as the four stages in the *software life cycle*. We will say a bit more about the first three stages later in this chapter when we specify, design and implement a new program, a cash register. But first, in the next section, we discuss a key issue in the management of the software development process.
Study Questions

11.1.1. What are the four stages of software development?

11.1.2. What is the interface of a software component?

11.1.3. What is integration testing?

11.1.4. What is the main advantage of integration testing?

11.2 The Software Development Process

By the time we are done creating a program, we will have necessarily specified, designed and implemented it. A key question is how to synchronize these activities. A number of approaches are possible.

One is to simply do everything all at once. Essentially, start coding right away, and specify and design the program as you code. This may work reasonably well for small programs but this approach is problematic with large programs. For example, it makes it difficult to split the work among several programmers: how can one programmer implement a component and another one simultaneously write code that uses that component unless the two programmers first agree on what the component does and how it should be used (specification, including interface)?

An approach that addresses these problems is to carry out the specification, design and implementation of a program in sequence. This is usually referred to as the waterfall model of software development. The idea is that the product of each stage flows as input into the next stage: a specification flows from the specification stage into the design stage, a design document flows from the design stage into the implementation stage, and code flows the implementation stage into the maintenance stage.

This approach may seem ideal because each stage of the development of the software relies on complete information from the previous stage. However, it is difficult to specify and design a very large program without writing any code to experiment with the software, and verify that the specification and design make sense.

An alternative to the waterfall model is to proceed incrementally by specifying, designing and implementing successively more complete versions of the software. This is the approach we took with the pay calculator: we built a first version that performed no error-checking.
11.2. THE SOFTWARE DEVELOPMENT PROCESS

The main advantage of **incremental** (or **iterative**) development is that knowledge and experience gained in developing one version of the software can be used in developing the next one. For example, it is possible to get feedback from the client on early versions. Another advantage is that the creation of the entire software proceeds as a sequence of smaller, more manageable projects. And finishing a version of the software, even an incomplete one, is a satisfying experience that typically generates excitement and increased motivation.

Note that the specification and design of a particular version of the program does not need to be completely done before its implementation. Details that concern only one component, such as I/O format, can be left to the developer of that component. In other words, incremental development does not have to be a series of smaller waterfall development projects. A fair amount of flexibility is possible in the development of each version.

There is a lot of evidence that incremental development is superior to the single-pass waterfall model. For example, Craig Larman and Victor Basili [LB03] surveyed the history of incremental development and concluded that incremental development concepts "have been and are recommended practice by prominent software-engineering thought leaders of each decade, associated with many successful large projects, and recommended by standards boards."

Fred Brooks [Bro87], an early leader in the field of software engineering, described the essential challenges of software engineering and identified incremental development as one of three promising solutions. He points out that software developers used to think of writing programs (as if they were recipes) but that as program size increased, software developers realized it was more appropriate to think of building programs. Similarly, Brooks suggested that instead of building programs, we now think of growing them (by using incremental approaches).

Note that incremental development is a key ingredient in all the recent agile methods, such as extreme programming.

Much more can be said about the software development process. At Clarkson, this is done in a course such as CS350 *Software Design and Development*.

---

1Brooks was also the 1999 winner of the ACM's Turing Award, which is widely regarded as the “Nobel Prize” of computer science [TA].
Study Questions

11.2.1. What is a disadvantage of coding right away without first specifying and designing the software?

11.2.2. What is the waterfall model of software development?

11.2.3. What is incremental software development?

11.2.4. What are the benefits of incremental development?

11.3 Specification

As is typical, the initial description of the cash register program is vague and incomplete. Our first task is to produce a precise description of what the program must do.

One strategy for producing a program specification is to pretend that the program is running and then imagine using it in every possible way. This is sometimes called working through scenarios. The goal is to explore and specify every detail of the program’s behavior.

One key detail is the size of the data file containing the list of products. Why is this important? There are two basic ways in which we can deal with this data: we can leave it in the file or we can store it in the program’s variables. Typically, files are stored on the computer’s hard drive while variables are stored in the computer’s main memory. Accessing data in main memory is normally much faster than on the hard drive, but the amount of main memory available for storing a program’s variables is limited.

For this version of the program, we will assume that the computers on which this program will run will have enough main memory to store all the product data. This assumption seems reasonable based on the fact that supermarkets carry an average of about 50,000 products and that each product should require about 38 bytes of storage. Even assuming 100,000 products, this adds up to less than 4 megabytes. In later versions of the program, we might consider removing this assumption and create a more general program.

A possible specification for the first version of the cash register program is shown in Figures 11.2 to 11.4. Notice how the assumptions about memory usage are clearly explained and documented.
A program that runs a cash register. It takes customer orders and computes the total cost of the order. The user enters product numbers and quantities. The program reads from a file containing a list of products each with a number, a unit price and a brief description.

Typical session:

(n)ew costumer  (a)dd to previous order  (c)lose register
? n

Enter product number and quantity (0 = end)

> 4237 1
  Bananas 1.25

> 5031 3
  3 @ 1.19
  Chickpeas 3.57

> 0

4 items Total: 4.82

(n)ew costumer  (a)dd to previous order  (c)lose register
?

The initial menu does not include add.

The command is entered as a single lowercase letter (n, a, c) or as a full name (new, add, close).

Figure 11.2: Specification of the register program (part 1 of 3)
The product number and quantity are integers. The product number is positive. The quantity can be negative but not zero. These numbers are entered on a single line and separated by spaces.

If a product with the given number cannot be found, a message is printed:

> 2347 3
Invalid product number. Please try again.

> 

The data file is called "products.txt" and is located in the same directory/folder as the program. The file contains a sequence of products. Each product consists of a number, a unit price and a text description. Those three pieces of data are given on a single line separated by a single space:

4237 1.25 Bananas
5031 1.19 Chickpeas
...

Products numbers are positive integers. Descriptions have no more than 30 characters.

Figure 11.3: Specification of the register program (part 2 of 3)
The computers on which this program will run will have enough main memory to store all the product data. (Later versions should reexamine this assumption.)

(According to the Food Marketing Institute, the average number of items carried in a supermarket in 2008 was 46,852 [1]. Each product requires an integer, a floating point number and a 30-byte description. On most 32-byte systems, this should require approximately 38 bytes [2]. Even assuming 100,000 products, this adds up to only 3.8 megabytes.)

No error-checking. (To be added in later versions.)

References:


Figure 11.4: Specification of the register program (part 3 of 3)
Study Questions

11.3.1. What does working through scenarios mean?

11.4 Design

As explained earlier in Section 11.1, to design a program means to identify its components, to choose or design algorithms and data structures for these components, and to write a precise specification for each component.

Recall that our goal is to design a modular program. There are three strategies that help us achieve this goal. First, we of course use abstraction, both procedural and data abstraction. Second, as we design the program, we make sure that components delegate as many tasks as possible to other components. Third, we aim for a design in which each component has its own secret. This typically leads to important aspects of the program being isolated from each other within different components.

Just like the specification of the program, the design can be carried out by working through scenarios, that is, by imagining that we are using the program in every possible way. The design then proceeds mainly as a sequence of what-who questions: What needs to be done? Who is going to do it? This is sometimes referred to as the what-who cycle. This design process is said to be responsibility-driven because it is driven by the identification and assignment of responsibilities.

Note that while designing a program, it is a good idea to delay deciding on minor details, especially those that involve only one component. Early on, we want to focus on the major aspects of the program without getting distracted (and possibly overwhelmed) by all the little details.

Figures 11.5 and 11.6 show a possible design for our cash register program. Four components are identified: a register that has overall control of the program and handles most of the user interaction, a user command (new, add or close), a product list that holds and provides access to the product data, and individual products.

The register component has overall control of the program but what it directly handles is only the interaction with the user; the rest is delegated to other components. For example, the storage of the products, as well as access to the file, is delegated to the product list. The register component also delegates part of the user interaction to the user command component.
COMPONENT: register

Responsibilities:

- runs the register

Collaborators: user command, product list, product

Implementation notes: handles the main user interaction and delegates the rest to other components

COMPONENT: user command

Responsibilities:

- represents a single user command
- reads from user
- says if command is close, new or add

Collaborators: none

Figure 11.5: Draft design of the cash register program (part 1 of 2)
COMPONENT: product list

Responsibilities:

- holds all the products
- reads the products from the file
- returns product with given number

Collaborators: product

Implementation notes: products are stored in a map of products indexed by product number

COMPONENT: product

Responsibilities:

- holds data for an individual product
- returns price, description or number
- reads data from file

Collaborators: none

Figure 11.6: Draft design of the cash register program (part 2 of 2)
At some point, the register component will read a product number from the user and need the price and description of that product. It will obtain that information from the product list. By including in our design a component for individual products, the product list will be able to return to the register an entire product instead of two separate pieces of information (the price and description). This will simplify the interaction between the register and the product list by keeping it a higher level, that is, at the level of entire products instead of at the level of the various elements of a product. In particular, this interaction will be independent from the exact contents of individual products.

For example, suppose that in a later version of the program, some products are taxable while others are not. Each product will have to indicate this and the register will need that information in order to correctly compute the cost of the order. The product component will need to be modified, as well as some aspects of the register component, but the interaction between the register and the product list will need not to change.

In addition, all that the product list needs to know about individual products is that they each have a number. The remaining contents of the products is irrelevant from the perspective of the product list. By including a product component in our design, we allow the product list to store the products as if they were closed boxes each labeled by a number. As a consequence, the product list is also independent from the exact contents of individual products.

Note that, as desired, each component of the program has its own secret. The register’s secret is the user interaction (except for the user commands). The user command’s secret is the reading and storage of those commands. The product list’s secret is the storage of the collection of products (without regards to the exact contents of those products). The product’s secret is the exact contents of an individual product.

We have now identified our program components and their responsibilities. The only significant data structure or algorithm issue in this program is the storage of the products. Since this only concerns the product list component, we could delay this decision until the implementation stage. Or we could decide that this is too important a decision so we should make it now so that the programmer in charge of implementing the product list is bound by this decision.

Let’s make the decision now. In the previous chapter, we discussed in detail the storage of the entries of the phone book program. We ended up settling on a map of entries indexed by name. The storage of the products is a somewhat similar situation. We have two basic options: a vector of products sorted by
product number, or a map of products indexed by product number.

There are two operations that need to be performed on the collection of products. First, the products must be read from the file and stored in whatever data structure we end up choosing. Second, given a product number, we need to find the corresponding product.

The products can be read and stored in a map in time $\Theta(n \log n)$, where $n$ is the number of products, by using the indexing operator. Note that the program specification does not say that the products in the file are in any particular order. Therefore, we can’t use the insert operation to speed this up to $\Theta(n)$, like we did in the case of the phone book. Searching for a product in a map takes $\Theta(\log n)$ time.

With a sorted vector option, we could fill the vector with the products and then sort the vector by product number. Later in these notes, we will see that this can be done in time $\Theta(n \log n)$. In addition, we can use the binary search algorithm to search for a product in a sorted vector in $\Theta(\log n)$ time.

So a map and a sorted vector would lead the register program to run in the same amount of time. But the map option is easier to implement. That’s because even though the STL includes both a sorting and a binary search algorithm (both of which we will learn to use later in these notes), the STL binary search algorithm can be used to search a sorted vector of products for a product, not for a product number. We could work around this problem but it would require extra coding. So we choose a map because that leads to a register program that’s just as fast but simpler to code.

All that is left to do to finish the design of the program is to write down precise specifications for each of the components. One possible outcome is shown in Figures 11.7 to 11.10.

Note how the ProductList reads the products from the file as soon as it is created. In addition, note how a Command returns information about its type through various $\text{is}$ methods.

Brooks (who we’ve mentioned earlier in this chapter) argues that one of the essential difficulties of software development is that software is “invisible and unvisualizable” [Bro87]. To address this difficulty, it is useful to create diagrams that help us to “see” our programs.

Figure 11.11 shows a diagram of the components of the cash register. Recall from earlier in these notes that in a component diagram, each component is represented by a box and an arrow from one component to another means that the first component uses the second. This particular diagram gives us a quick overview of the components of the program and how they depend on
COMPONENT: class Register

Represents an entire register.

Methods:

    void run()
        Runs the register. See program spec for details.

Global constant:

    const char kcsProductFileName[] = "products.txt"

Collaborators: ProductList, Product, Command

Implementation notes: Handles the main user interaction and delegates the rest to other components. Holds the product data in a ProductList that’s initialized when the Register is created.

Figure 11.7: The Register class
COMPONENT: class Command

Represents a user command.

Methods:

Command()
    Creates an invalid command.

bool is_new() const
bool is_add() const
bool is_close() const
    Return true if the command is of the corresponding type.

Operators:

    istream & operator>>( istream & in, Command & c )
    Reads a command from the user. Reads an entire line of
text containing one of the following: n, new, a, add, c,
close. No error-checking.

Collaborators: none

Figure 11.8: The Command class
COMPONENT: class ProductList

Holds and provides access to the list of products.

Methods:

ProductList( const string & product_file_name )
   Creates a product list initialized with products from
   the file with the given name. See program spec for a
   description of the file format. Closes the file when
   done. No error-checking.

   const Product * get_product( int nbr ) const
   Returns a pointer to the product with the given number.
   Returns NULL if the product is not found.

Collaborators: Product

Implementation notes: Products are stored in a map of Products
indexed by product number.

Figure 11.9: The ProductList class
COMPONENT: class Product

Represents an individual product.

Methods:

Product()
    Creates an invalid product.

int number() const
float price() const
const string & description() const
    Returns the product number, price or description.

Operators:

istream & operator>>( istream & in, Product & p )
    Reads a product from stream in. Reads an entire line of text containing the product number, price and description. See program spec for description of format. No error-checking.

Collaborators: none

Figure 11.10: The Product class
11.4. DESIGN

Figure 11.11: A high-level component diagram

Figure 11.12 shows a more detailed component diagram that indicates the main methods and data members of each component. A horizontal line separates the public members (top) from the private ones (bottom).

Components diagrams are useful but they’re static: they show what the components are but they don’t show them in action. In contrast, Figure 11.13 shows an interaction diagram. In an interaction diagram, each component is represented by a vertical line and a labeled arrow from one component to another represents a message being sent from the first component to the second one. Unlabeled arrows represent control returning to the first component. A component diagram is like a photograph of the program; an interaction diagram is more like a video.

Interaction diagrams usually include details that are normally determined during the implementation of a program. For example, the diagram of Figure 11.13 indicates that price gets called before description. This minor detail will be decided during the implementation of the Register. One approach is to draw a tentative diagram at the design stage and then revise it after the implementation of the software is complete. Another approach is
Figure 11.12: A more detailed component diagram
Figure 11.13: An interaction diagram
to consider interaction diagrams as tentative outlines of interaction and leave
them as that.

A single interaction diagram may be able to describe the entire execution
of a very small program. For larger programs, multiple diagrams are needed,
with each diagram describing a separate scenario.

**Study Questions**

11.4.1. Why should we aim for a design in which each component has its own
secret?

11.4.2. What is the what-who cycle?

11.4.3. What is responsibility-driven design?

11.4.4. What are two benefits of including in the design of the cash register
program a component for individual products?

**11.5 Implementation**

As a single-person project, the cash register can be implemented according to
the following plan:


3. Code and test Command.

4. Code Register and test in combination with the other three compo-
nents.

The implementation of the Command and Product components is fairly
straightforward. It is available on the course web site under Register1, to-
gether with complete documentation (specification, design, diagrams).

Figure 11.14 shows the declaration of the ProductList class. The type
ProductMap is defined to make the code more readable and easier to change.
This also reduces the amount of typing we need to do.
class ProductList
{
    public:
    ProductList( const string & product_file_name );

    const Product * get_product( int nbr ) const {
        ProductMap::const_iterator itr = m_products_.find( nbr );
        if ( itr == m_products_.end() )
            return NULL;
        else
            return &(itr->second);
    }

    private:
    typedef map<int, Product> ProductMap;

    ProductMap m_products_; // products indexed by number
};

Figure 11.14: The declaration of ProductList

ProductList::ProductList( const string & product_file_name )
{
    ifstream f( product_file_name.c_str() );

    Product product;
    while ( f >> product ) {
        m_products_[ product.number() ] = product;
    }
}

Figure 11.15: The default constructor of ProductList
for ( ProductMap::const_iterator itr = m_products_.begin();
    itr != m_products_.end();
    ++itr ) {
    cout << " " << itr->first << ' '
        << itr->second.price() << ' '
        << itr->second.description() << endl;
}

Figure 11.16: Displaying the contents of the product map

Figure 11.15 shows an implementation of the default constructor of ProductList. Note how the indexing operator is used solely for the purpose of adding new elements to the map.

For testing purposes, it is convenient to temporarily include in the constructor code that prints the contents of the map, as shown in Figure 11.16.

Figure 11.14 also shows the implementation of the ProductList method get_product. Note how the address-of operator & is used to obtain a pointer to the map element itr->second. Also note that because get_product is declared constant, within the body of the method, every data member of the ProductList receiver, including m_products_, is considered constant, which implies that the constant version of find will be used, and that version returns a constant iterator. That’s the reason why itr must be declared a constant iterator.

Figure 11.17 shows the declaration of the Register class. Note the introduction of four (private) helper methods that will be used by run. These methods help simplify the implementation of run. They are introduced during the implementation of run and are not part of the overall design of the program. This illustrates how the implementation stage can still involve some amount of design.

As mentioned earlier, the full implementation of the cash register, as well as complete documentation, is available on the course web site under Register1.

Exercises

11.5.1. Suppose that a file contains various products, as in the cash register program. Write code segments that perform each of the following:

a) Read the file and store all the products in a map indexed by product
const string kcsProductFileName = "products.txt";

class Register
{
public:
    Register() : product_list_(kcsProductFileName) {}

    void run();

private:
    ProductList product_list_;

    void process_order(int & num_items, float & cost) const;
    // Takes an order. In the program spec, this goes from
    // right after the user enters a new or add command, all the
    // way until he or she enters 0 and the totals are printed.

    void print_product_info(const Product * p, int quantity) const;
    // In the program spec, this is the information printed
    // after the user has entered the product number and
    // quantity.

    void print_totals(int num_items, const float & cost) const;
    // In the program spec, this occurs right after the user an
    // entered 0 to terminate an order.

    void print_money(const double & amount, int width) const
    // Prints an amount of money right-justified in a block of
    // characters of the given width. For example,
    // printMoney(1.42, 5) prints " 1.42".
    {
        cout << fixed << setprecision(2)
        << setw(width) << right << amount;
    }
};

Figure 11.17: The declaration of Register
b) Print all the products in the map that cost less than $1.

c) Create another map that contains all the products that cost less than $1. That second map should be indexed by product \textit{number}.

d) Print all the products in reverse alphabetical order of description. Use constant reverse iterators.

11.5.2. Modify the cash register program so that some products are taxable while others are not. The program should display not only the total for the bill but also the number of taxable items bought and the tax charged on them. Use 7\% for the tax.

Based on the above description, revise the specification of the program. In particular, this involves modifying the format of the product data file. Then revise the design and implementation of the program. Make sure you test any modified components in isolation before integrating them into the rest of the program.
Chapter 12

Generic Algorithms

In these notes, we have used function templates to create generic functions that work on various types of arguments. In a similar way, we have used class templates to implement data structures that store various types of elements. In this chapter, we will push the idea of generic programming one step further and learn to implement generic algorithms that work on various data structures. Iterators and functions templates will play a key role. We will also learn about some of the generic algorithms available in the STL.

12.1 A Generic Counting Algorithm

Consider the following computational problem: you’re given a list \( \text{ls} \) and an element \( \text{e} \) and you need to count the number of occurrences of \( \text{e} \) in \( \text{ls} \). There is a simple algorithm that solves this problem: start with a counter set to 0, then scan the list from beginning to end, adding 1 to the counter every time you see an element equal to \( \text{e} \).

Note that this problem is generic in that it makes sense for lists that store any type of element. The algorithm too is generic: it can be used on lists that store any type of element.

As we learned earlier in these notes, we can use function templates to implement this algorithm as a single generic function that can be used on lists of any type. A generic function that implements the generic counting algorithm is shown in Figure 12.1.

There are a number of advantages to being able to write a single function that works on lists of any type instead of having to write a separate function
template <class T>
int count( const list<T> & ls, const T & e )
// Assumption on T: values of type T can be compared using ==.
{
    int count = 0;
    for ( typename list<T>::const_iterator itr = ls.begin(); itr != ls.end(); ++itr )
    {
        if ( *itr == e ) ++count;
    }
    return count;
}

Figure 12.1: A generic function that counts elements in a list

for every type of list:

1. There is less code to write (obviously).

2. It is easier to maintain the code because, when needed, only one function
   needs to be changed.

3. It is possible to put the function in a library. On the other hand, there
   is no way we can create a library that includes a function to print every
   possible type of list because there is no limit on the number of different
   types of lists programmers can create.

Now suppose that we are interested in counting the number of occurrences
of a character in a vector. This is essentially the same problem and it can be
solved by essentially the same algorithm: scan the vector from beginning to
end while counting the number of occurrences of the given element. A function
that implements the algorithm for vectors is shown in Figure 12.2.

Now, what would be nice is if we could have a single generic function that
can count in either lists or vectors. For that to be possible, we need a uniform
mechanism for accessing elements in either a list or a vector.

As mentioned earlier, indices would be an inefficient way of accessing el-
ements in a list. A better solution is to add iterators to vectors and this is
precisely what the designers of the STL did.

Figure 12.3 shows a generic function that can count in both lists and vec-
tors. Note that this function can actually do much more: it can count elements
12.1. A GENERIC COUNTING ALGORITHM

```cpp
template <class T>
int count( const vector<T> & v, const T & e )
// Assumption on T: values of type T can be compared using ==.
{
    int count = 0;
    int n = v.size();
    for ( int i = 0; i < n; ++i ) if ( v[i] == e ) ++count;
    return count;
}
```

Figure 12.2: A generic function that counts elements in a vector

in any container that provides constant iterators and constant versions of the methods begin and end. For example, this includes C++ strings (the class string).

Now, what about arrays? The generic counting algorithm can be applied to arrays and an implementation for arrays is given in Figure 12.4.

We would of course like to push the idea of generic programming further and create a single function that counts in any type of container including arrays. The function of Figure 12.3 won’t work because arrays are not a class so they don’t support the begin and end methods.

But arrays have iterators: pointers to elements. We already know that pointers can be dereferenced (*) to access what they point to. But pointers can also be compared by using the equality and inequality testing operators (==, !=). In addition, pointers can be incremented: if p points to an array element, ++p will cause the pointer to point to the next element.

Therefore, all that we need in order to create a function that works on lists, vectors and arrays is a way to avoid using the begin and end methods. In the function of Figure 12.3, these methods are used to obtain iterators that indicate where the counting should start and stop. A solution is to provide those start and stop iterators as arguments to the function, as shown in Figure 12.5.

To count in a list ls, we would call the function like this:

```cpp
my_count(ls.begin(), ls.end(), e)
```

To count in an array a of size n, we would write

```cpp
my_count(a, a+n, e)
```
template <class Container, class T>
int count( const Container & c, const T & e )
// Assumption on Container: it is a class with constant
// iterators as well as constant begin and end methods.
// Assumption on T: values of type T can be compared using ==.
{
    int count = 0;
    for (typename Container::const_iterator itr = c.begin(); itr != c.end(); ++itr ) {
        if ( *itr == e ) ++count;
    }
    return count;
}

Figure 12.3: A generic function that counts elements in virtually any type of container

template <class T>
int count( const T * a, int n, const T & e )
// Assumption: a points to an array of size at least n.
// Assumption on T: values of type T can be compared using ==.
{
    int count = 0;
    for ( int i = 0; i < n; ++i ) if ( a[i] == e ) ++count;
    return count;
}

Figure 12.4: A generic function that counts elements in an array
12.1. A GENERIC COUNTING ALGORITHM

```cpp
template <class Iterator, class T>
int my_count( Iterator start, Iterator stop, const T & e )
// Assumption: [start, stop) is a valid range.
// Assumption on Iterator: iterators point to values of type T.
// Assumption on T: values of type T can be compared using ==.
{
    int count = 0;
    for ( Iterator itr = start; itr != stop; ++itr ) {
        if ( *itr == e ) ++count;
    }
    return count;
}
```

Figure 12.5: A generic function that counts elements in any range of positions

This works because, as we know, when needed, an array name is automatically converted to a pointer to the first element of the array. And a+n points n elements to the right, which is just past the last element of the array.

We now have a generic function that can count elements in any range of positions specified by two iterators. Note that the range includes the start iterator but not the stop iterator. In other words, the range is the interval [start, stop).

With these arguments, the function is a little less convenient to use when counting over an entire container. For example, to count in a list ls with the earlier generic function (Figure 12.1), all we had to write was

```
count(ls, e)
```

On the other hand, the function is now more flexible since it allows us to count over any portion of a container, not just the entire container.

Note that we called our last generic function my_count instead of count. This is to avoid a conflict with a library function count that takes the same arguments. We will see later in this chapter that the STL includes implementations of generic algorithms that solve a wide variety of common problems.

Source code for the generic functions of this section is available on the course web site under generic_count.cpp.
Study Questions

12.1.1. What is a generic algorithm?

12.1.2. What are three distinct benefits of implementing a generic algorithm as a single generic function?

12.1.3. What role do iterators play in the implementation of generic algorithms?

Exercises

12.1.4. Write a generic function

\[ \text{my\_fill}(\text{start}, \text{stop}, e) \]

that sets to \( e \) every element in the range \([\text{start}, \text{stop})\).

12.1.5. Write a generic function

\[ \text{my\_replace}(\text{start}, \text{stop}, x, y) \]

that replaces by \( y \) every occurrence of element \( x \) in the range \([\text{start}, \text{stop})\).

12.1.6. Write a generic function

\[ \text{my\_find}(\text{start}, \text{stop}, e) \]

that returns an iterator to the first occurrence of element \( e \) in the range \([\text{start}, \text{stop})\).

12.2 Iterators

Iterators provide a uniform mechanism for accessing elements in a wide variety of containers. This is one of the key tools that allow us to implement generic algorithms that work on a wide variety of data structures. For example, in the last section, we implemented a generic algorithm that counts elements in lists, vectors and arrays. In this section, we learn some additional important details about iterators.
12.2. ITERATORS

template <class Iterator>
void print( Iterator start, Iterator stop )
// Assumption on Iterator: iterators that point to values that
// can be printed using the output operator (<<).
{
    for ( Iterator itr = start; itr != stop; ++itr ) {
        cout << *itr << ' ';
    }
    cout << endl;
}

Figure 12.6: A generic function that prints a range of elements

We already know that STL lists have two types of iterators: plain iterators (iterator) and constant iterators (const_iterator). Vectors and strings also provide those two types of iterators. But both of these containers, as well as almost every other type of STL container, provide two additional iterator types: reverse iterators (reverse_iterator) and constant reverse iterators (const_reverse_iterator).

Reverse iterators, as the name indicates, are designed to traverse a container in reverse order. Containers that support these iterators also provide a method rbegin (reverse begin) that returns a reverse iterator that points to the last element of the container and a method rend (reverse end) that returns a reverse iterator that points to a position just before the first element of the container. In addition, the ++ and — operators work in reverse direction with reverse iterators.

For example, Figure 12.6 shows a generic function that prints all the elements in a range. When given ordinary iterators, as in

    print( ls.begin(), ls.end() );

the function prints in the usual order. When given reverse iterators, as in

    print( ls.rbegin(), ls.rend() );

the function prints in reverse order.

So iterators can be of four different types (plain, constant, reverse, constant reverse). But there are several iterator categories. The two most important ones are bidirectional iterators and random-access iterators.
CHAPTER 12. GENERIC ALGORITHMS

Table 12.1: Operations supported by bidirectional iterators

<table>
<thead>
<tr>
<th>Operation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>*itr</td>
<td>Returns a reference to the element that *itr points to.</td>
</tr>
<tr>
<td>itr-&gt;f(args)</td>
<td>Returns a reference to the element that *itr points to and either sends message f with arguments args to it or returns a reference to data member var. The element must be an object or a structure.</td>
</tr>
<tr>
<td>itr-&gt;var</td>
<td></td>
</tr>
<tr>
<td>++itr</td>
<td>Makes *itr point to the next or previous element.</td>
</tr>
<tr>
<td>--itr</td>
<td></td>
</tr>
<tr>
<td>itr1 == itr2</td>
<td>Returns true if itr1 and itr2 point or don’t point to the same element.</td>
</tr>
<tr>
<td>itr1 != itr2</td>
<td></td>
</tr>
</tbody>
</table>

The difference between these iterator categories is in the set of operations that they support. Bidirectional iterators support the basic operations that we are already familiar with: *, ->, ++, --, == and !=. These operations are described in Table 12.1.

Random-access iterators support all the operations supported by bidirectional iterators plus the following ones: +i, -i, -, < and [i]. These operations are described in Table 12.2. Note that itr[0] is equivalent to *itr and that, in general, itr[i] is equivalent to *(itr + i).

List and map iterators are typical examples of bidirectional iterators. Vector and string iterators, as well as pointers to array elements, are typical examples of random-access iterators.

Study Questions

12.2.1. What element does a reverse begin iterator point to? What about a reverse end iterator?

12.2.2. In what direction does the — operator move a reverse iterator?
itr + i
itr − i

Returns an iterator that points i elements to the right or left of itr.

itr1 − itr2

Returns the distance, in elements, between itr1 and itr2.

itr1 < itr2

Returns true if itr1 points to the left of itr2.

itr[i]

Accesses the element i positions to the right of the element that itr points to.

Table 12.2: Additional operations supported by random-access iterators

12.2.3. What operations are supported by bidirectional iterators?

12.2.4. What operations are supported by random-access iterators but not by bidirectional iterators?

Exercises

12.2.5. Experiment with the operations of bidirectional iterators by writing a test driver that uses, on list iterators, all the operations shown in Table 12.1.

12.2.6. Experiment with the operations of random-access iterators by writing a test driver that uses, on pointers to array elements, all the operations shown in Table 12.2.

12.3 Generic Algorithms in the STL

The STL includes a large number of generic functions that implement generic algorithms for a wide variety of common tasks. The functions themselves are commonly referred to as generic algorithms. This section will provide a brief overview of STL generic algorithms. Several additional algorithms are described in a reference such as cplusplus.com [CPP].
Table 12.3 lists some generic algorithms that are designed to be used on sequences such as lists, vectors and arrays. Recall that ranges of positions such as \([\text{start}, \text{stop}]\) are specified by iterators.

The \texttt{find} generic algorithm performs a sequential search, which implies that it runs in linear time. The STL includes several versions of the binary search algorithm we discussed earlier in these notes. Three of them are shown in Table 12.4. These algorithms run in logarithmic time when they are given random-access iterators.

Sorting is an important and frequent operation on sequences of elements in large part precisely because it allows for faster searches. The STL includes a generic sorting algorithm that is also shown in Table 12.4. This algorithm is very efficient but requires that the range be specified by random-access iterators. Therefore, the algorithm can be used on arrays and vectors, but not on lists. For this reason, the class \texttt{list} includes a method \texttt{sort()} that uses a different sorting algorithm.

Table 12.5 lists some other useful generic algorithms that are typically used on non-container arguments. This table also includes generic algorithms that operate on iterators. These algorithms provide the same functionality as the + and − operations of random-access iterators but they can also be used on bidirectional iterators. When used with random-access iterators, these algorithms run in constant time. When used with bidirectional iterators, the algorithms run in linear time.

Some of the STL generic algorithms perform very simple tasks. In fact, some of them, including most of the algorithms presented in this section, are typical of examples and exercises given in a first programming course. So you may wonder why it is useful to include them in a library. You may also wonder if it’s worth your time and effort to learn how to use these algorithms.

In general, using software components from a standard library instead of implementing your own has several advantages. First, in the long run, the time invested in learning how to use these library components is more made up by the time saved in implementing them over and over again. This is clearly true in the case of more complex components such as lists, vectors, maps and strings, but it is also true even for simple components such as the STL generic algorithms presented in this section.

The second advantage is that programs that use library components are usually more reliable because these components have typically been tested by many other users.

Finally, programs that use standard components are easier to understand
12.3. GENERIC ALGORITHMS IN THE STL

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>count( start, stop, e )</td>
<td>Returns the number of occurrences of element e in the range [start, stop). Uses == on elements.</td>
</tr>
<tr>
<td>find( start, stop, e )</td>
<td>Returns an iterator to the first occurrence of element e in the range [start, stop). Returns stop if e is not found. Uses == on elements.</td>
</tr>
<tr>
<td>max_element( start, stop )</td>
<td>Returns an iterator that points to the maximum or minimum element in the range [start, stop). Uses &lt; on elements.</td>
</tr>
<tr>
<td>min_element( start, stop )</td>
<td></td>
</tr>
<tr>
<td>copy( start, stop, dest_begin )</td>
<td>Copies the elements in the range [start, stop) to a range of positions that begins at dest_begin. If n is the number of elements that are copied, returns an iterator that points n positions to the right of dest_begin. Copies forward, from start to stop. Typically not useful when dest_begin falls within the range [start, stop).</td>
</tr>
<tr>
<td>copy_backward( start, stop, dest_end )</td>
<td>Copies the elements in the range [start, stop) to a range of positions immediately to the left of dest_end. If n is the number of elements that are copied, returns an iterator that points n positions to the left of dest_end. Copies backward, from stop to start. Use instead of copy when the destination range begins within [start, stop).</td>
</tr>
<tr>
<td>fill( start, stop, e )</td>
<td>Sets all the elements in the range [start, stop) to be copies of element e.</td>
</tr>
<tr>
<td>replace( start, stop, x, y )</td>
<td>Replaces all occurrences of element x by a copy of element y in the range [start, stop). Uses == on elements.</td>
</tr>
<tr>
<td>reverse( start, stop )</td>
<td>Reverses the order of the elements in the range [start, stop).</td>
</tr>
</tbody>
</table>

Table 12.3: Some generic sequence algorithms
CHAPTER 12. GENERIC ALGORITHMS

binary_search( start, stop, e )
lower_bound( start, stop, e )
upper_bound( start, stop, e )

Performs a binary search in the range [start,stop). Assumes that the range is sorted with respect to <. Uses < on elements. The first version returns true if element e is present in the range. Otherwise, it returns false. The second version returns an iterator that points to the first position where e could be inserted in the range while preserving the order. The third version returns an iterator that points to the last such position.

sort( start, stop )

Sorts the elements in the range [start,stop) using the < operator. Requires random-access iterators.

Table 12.4: Some generic algorithms for searching and sorting

swap(x, y)

Swaps the values of x and y.

max(x, y)
min(x, y)

Returns the maximum or minimum of x and y. Uses < on the arguments.

advance( itr, n )

Advances itr by n positions. n can be negative. Uses + or − once if the iterator is random-access. Otherwise, uses ++ or −− repeatedly.

distance( itr1, itr2 )

Returns the distance from itr1 to itr2, in elements. Uses − once if the iterators are random-access. Otherwise, uses ++ or −− on itr1 until it reaches itr2. Assumes that itr2 is reachable from itr1.

Table 12.5: Some other generic algorithms
because other developers are already familiar with these components.

The use of software components from a standard library is an example of software reuse. Reusing any software component, and not just library components, generally has the same three benefits: faster development, increased reliability, and software that’s easier to understand.

The iterator generic algorithms are included in the library file iterator. The other STL generic algorithms are in the library file algorithm. All of them are part of the std namespace.

**Study Questions**

12.3.1. What are three benefits of using standard software components such as the STL generic algorithms?

**Exercises**

12.3.2. Why is the generic algorithm copy typically not useful when dest.begin falls within the range [start,stop)?

12.3.3. Implement your own version of the following generic algorithms. Use the prefix my on each of them (as in my_count) to avoid conflicts with the corresponding STL algorithms.

   a) swap.
   b) max.
   c) max_element.
   d) copy.
   e) copy_backward.
   f) reverse. *Hint*: Make sure you consider sequences of even and odd length.

12.3.4. Experiment with the STL searching and sorting algorithms by writing a test driver that uses all the algorithms shown in Table 12.4 as well as the sort method of class list. To see the difference between lower_bound and upper_bound, make sure you search for an element that occurs multiple times in the sequence.
Chapter 13

Implementation of Linked Lists

In this chapter, we will learn how to implement a basic class of linked lists. Dynamic memory allocation and pointers will play a key role. The techniques we learn in this chapter are the basis for the implementation of other important linked data structures, such as the trees that are typically used to implement STL maps.

13.1 Nodes and Links

We start by addressing the basic setup of our class of linked lists. The elements of a linked list are supposed to be scattered in the computer’s memory and somehow linked together. The standard way of achieving this is to store each element together with pointers to the next and previous elements in the list. More precisely, we will store each element in a node that will combine the element, a pointer to the node containing next element and a pointer to the node containing previous element.

Figure 13.1 shows a class ListNode that implements this idea. Because lists are generic, we need to be able to create nodes that can store any type of element. This implies that nodes must be generic too. This is why ListNode is a class template. Its type parameter T represents the type of element stored in the list.

List nodes are meant to be used only in the implementation of our class of linked lists, which we will call List to distinguish from the STL class list. It’s safer to prevent other software components from using these nodes. This is why we declare the data members private and grant friendship only to List.
template <class T>
class ListNode
// T is the type of element stored in the list.
{
    friend class List<T>;

private:
    T element;
    ListNode<T> * next;
    ListNode<T> * previous;
};

Figure 13.1: A class of nodes

Each List object needs to hold one node for each element in the linked list. Where should those nodes be declared? The only way to declare a large number of nodes inside a List object is to put them in an array or vector that’s a data member of the List object. But eventually, this array or vector would need to grow and this would require copying every existing node from the current array or vector to the new larger one. This would take linear time, making it impossible to implement operations like push_back, push_front and insert in constant time.

The solution is to store the nodes outside the List object. This can be done by dynamically allocating the nodes, the same way we dynamically allocated arrays earlier in these notes.

Even though the nodes of our linked lists will not be stored inside the List objects, each List object will need to be able to somehow hold its nodes. One way is to have each List object contain a pointer to the first node in the list, as shown in Figure 13.2. Once again, note the use of the prefix p to make it clear that the data member holds a pointer to the head node and not the head node itself.

The head node will not contain the first element of the list. Instead, the head node will be a dummy head node: this node will not be used to store a list element, it will be there only to ensure that every node has a predecessor. This is a standard technique that eliminates special cases when implementing some of the list operations.

We will also have the next pointer of the last node of the list point to the
template <class T>
class List
// T is the type of element stored in the List.
{
  private:
    ListNode<T> * p_head_node;
};

Figure 13.2: The class List with a pointer to its head node

dummy head node, and the previous pointer of the dummy head node point
the last node. This will make the list circular and ensure that every node,
including the last one, has a successor. More special cases will be eliminated
this way.

The kind of linked list we have just described is called a circular doubly-
linked list with dummy head node. The list is said to be doubly linked because
each node is linked to both its successor and predecessor.

Study Questions

13.1.1. Why can’t we store a list’s nodes inside the list object?

13.1.2. What is a dummy head node?

13.1.3. What is the advantage of having a dummy head node and making the
list circular?

13.2 Some Basic Methods

We are now ready to start adding operations to our class List. These opera-
tions will form a small but representative subset of the STL list operations.
You will be asked to implement additional operations in the exercises. Note
that some features of STL lists will be omitted. For example, our lists will
not have constant iterators. This will limit our ability to work with constant
versions of our lists.

The implementation of List will be done gradually. In this section, we
start with a default constructor, push_back, pop_back and back. Figure 13.3
template <class T>
class List
    // T is the type of element stored in the list.
{
public:
    List() {
        p_head_node = new ListNode<T>;
        p_head_node->next = p_head_node->previous = p_head_node;
    }
    
    T & back() { return p_head_node->previous->element; } 
    const T & back() const { 
        return p_head_node->previous->element;
    }
    
    void push_back( const T & new_element);
    void pop_back();

    void test_print() const; // for testing only
private:
    ListNode<T> * p_head_node;
};

Figure 13.3: A first version of List

shows a class declaration that includes those methods as well as a method test_print that prints the contents of the list. This method is what we can call an internal test driver: it is there only for testing purposes. It will no longer be needed once we implement iterators. And it should be removed (or commented out) before the final version of the class is produced.

The implementation of test_print is shown in Figure 13.4. The method traverses the linked list by using a pointer pNode that is initialized to point to the node that contains the first element (the node that follows the dummy head node) and then travels down the list until it reaches the dummy head node.

Note how the next pointer of the dummy head node is accessed: p_head_node->next. This uses the same dereference-and-select operator that
### 13.2. SOME BASIC METHODS

```cpp
template <class T>
void List<T>::test_print() const {
    for ( ListNode<T> * p_node = p_head_node->next;
        p_node != p_head_node;
        p_node = p_node->next ) {
        cout << p_node->element << ' ';
    }
    cout << endl;
}
```

Figure 13.4: The internal test driver `test_print`

was introduced in the context of map iterators in Chapter 10. Recall that this is equivalent to

```cpp
(*p_head_node).next
```

where `*p_head_node` refers to the node that `p_head_node` points to.

Note also how the pointer is made to point to the next node: `p_node = p_node->next`. This plays the same role here as adding one to an array index, or incrementing an iterator to make it move forward.

The default constructor has to create an empty list. An empty circular doubly-linked list with dummy head node consists of just a dummy head node whose next and previous pointers point to the dummy head node itself. The default constructor must allocate the dummy head node and set the pointers.

Figure 13.3 includes the implementation of the default constructor. The first line allocates a node and stores its address in the head pointer. The second line access the `next` and `previous` pointers of the head node and sets them to `p_head_node`.

Note that to access the next and previous pointers of the dummy head node, the default constructor needs access to the private data members of `ListNode`. This is one instance where `List` needs to be a friend of `ListNode`.

Figure 13.3 also includes the implementation of two versions of the `back` method. The constant version will be chosen automatically be the compiler for use on constant lists: it returns a constant reference to prevent the user from modifying the list. The non-constant version of `back` will be used on non-constant lists: it returns a plain reference that allows the user to modify the last element of the list.
template <class T>
inline void List<T>::push_back( const T & new_element ) {
    // set a pointer to the last node
    ListNode<T> * p_last_node = p_head_node->previous;

    // create new node and set its contents
    ListNode<T> * p_new_node = new ListNode<T>;
    p_new_node->element = new_element;
    p_new_node->next = p_head_node;
    p_new_node->previous = p_last_node;

    // finish linking new node to list
    p_last_node->next = p_new_node;
    p_head_node->previous = p_new_node;
}

Figure 13.5: The push_back method

Implementations of push_back and pop_back are shown in Figures 13.5 and 13.6. The comments describe what these methods do. To follow this kind of code, it is very useful to draw pictures that show the nodes and the pointers that link those notes. Drawing such pictures is also very useful when writing the code.

Complete source code for this first version of our class List is available on the course web site under List1.0.

Exercises

13.2.1. Add the methods front, push_front and pop_front to the class List.

13.2.2. Add size and empty. To implement these methods efficiently, add a new integer data member to the class to keep track of the current size of the list. Don’t forget to revise the other methods to initialize and then update this data member.

13.2.3. Add the constructors List(n) and List(n, e). Hint: Use push_back repeatedly. Don’t forget to first initialize the list to be empty.
13.3. ITERATORS, INSERT AND ERASE

Figure 13.6: The pop_back method

13.2.4. Repeat the previous exercise but this time implement each of the constructors without using either push_back or push_front. Do this carefully, so you end up saving two pointer assignments for each element added to the list. Hint: There is no need to link the last node with the head node until the very end.

13.2.5. Add swap. Make sure your implementation runs in constant time.

13.3 Iterators, Insert and Erase

We now add iterators to our class of linked lists. This will require some thought.

First, list iterators cannot simply be pointers to elements. If they were, then dereferencing an iterator (by using the * operator) would give access to the element that the iterator points. But incrementing the iterator would result in an iterator that points to the next memory location and this is certainly not where the next element is located because elements are contained in nodes that also contain the next and previous pointers.

Second, list iterators cannot simply be pointers to nodes either. In that case, dereferencing an iterator would not give access to an element but to the
node that contains that element. In addition, incrementing such an iterator would result in an iterator that points to the next memory location, which is most likely not where the next node is stored.

Therefore, list iterators will have to be objects that we define ourselves. From the user’s perspective, a list iterator points to an element. We will implement this by having the iterator hold a pointer to the node that contains that element.

Figure 13.7 shows the declaration of our class of iterators. This declaration includes the implementation of the default constructor, the dereferencing operator (*) and the inequality testing operator (!=).

Figure 13.8 shows the implementation of the prefix ++ operator. The operator needs to return the receiver so that code such as

\[
i_tr1 = ++i_tr2;
\]

can work properly. Recall that in a method, the variable \texttt{this} always points to the receiver.

Note how in Figure 13.7, a dummy integer argument is used to distinguish between the prefix and postfix versions of the ++ and --- operators. The implementation of the postfix ++ operator is shown in Figure 13.9.

Most iterator operations need to access the three ListNode members \texttt{element}, \texttt{next} and \texttt{previous}. A convenient way of making this possible is to have ListNode grant friendship to ListIterator. Even though this increases dependence between the two classes, it makes sense here since these two classes are really part of the same “package”.

To complete the addition of iterators to our class, we need to define the iterator type and implement the methods \texttt{begin} and \texttt{end}. Recall that the users of STL lists expect that list iterators will be of type \texttt{list<T>::iterator}. Therefore, we include in our class \texttt{List} the following declaration:

\[
\textbf{typedef} \texttt{ListIterator<T> \ iterator;}
\]

This will cause \texttt{List<T>::iterator} to mean \texttt{ListIterator<T>}. The \texttt{begin} and \texttt{end} methods can be implemented as follows:

\[
\text{iterator begin()} \{ \textbf{return} \texttt{iterator( p\_head\_node->next );} \} \\
\text{iterator end()} \{ \textbf{return} \texttt{iterator( p\_head\_node );} \}
\]
template <class T>
class ListIterator
// T is the type of element stored in the List.
{
    friend class List<T>;

public:
    ListIterator() { p_current_node = NULL; }
    T & operator*() { return p_current_node->element; }
    bool operator!=(const ListIterator & rhs) const {
        return (p_current_node != rhs.p_current_node);
    }
    ListIterator & operator++(); // prefix version (++itr)
    ListIterator & operator--();
    ListIterator operator++(int); // postfix version (itr++)
    ListIterator operator--(int);

private:
    ListIterator( ListNode<T> * p ) { p_current_node = p; }
    ListNode<T> * p_current_node; // points to the node that
    // contains the element that the iterator currently
    // points to
};

Figure 13.7: The declaration of ListIterator
template <class T>
inline ListIterator<T> & ListIterator<T>::operator++()
// prefix version (++itr)
{
    p_current_node = p_current_node->next;
    return *this;
}

Figure 13.8: The prefix ++ operator of class ListIterator

template <class T>
inline ListIterator<T> ListIterator<T>::operator++(int)
// postfix version (itr++)
{
    // copy the current value of the iterator
    ListIterator<T> original_itr = *this;
    // increment iterator
    p_current_node = p_current_node->next;
    // return original
    return original_itr;
}

Figure 13.9: The postfix ++ operator of class ListIterator
template <typename T>
void print( List<T> & ls )
// Assumption on T: values can be printed using <<.
{
    for ( typename List<T>::iterator itr = ls.begin();
        itr != ls.end();
        ++itr ) {
        cout << *itr << ' ';
    }
    cout << endl;
}

Figure 13.10: A function that prints a List

In each of these methods, a call is made to the second (private) constructor of class ListIterator. This creates an iterator and initializes it to contain the appropriate pointer. That iterator is then returned. Note that access to this private constructor of ListIterator is possible from within the begin and end methods of the class List because we made List a friend of ListIterator (see Figure 13.7).

Now that we have iterators, we can add a few more methods to our class List. But first, let's write a print function that can be used for testing, as shown in Figure 13.10. This allows us to remove the test_print method from List. Note that the argument of print really should be passed by constant reference, but we can't do this right now because we haven't implemented constant iterators.

Figures 13.11 and 13.12 show the implementation the insert and erase methods. These implementations use the fact that List can access the private members of both ListNode and ListIterator. Once again, the code of these methods are easier to follow by using pictures.

Source code for a second version of List that includes iterators and the insert and erase methods is available on the course web site under List1.1.

Study Questions

13.3.1. Why does the iterator operator ++ return its receiver?
template <class T>
inline ListIterator<T> List<T>::insert( iterator itr, const T & new_element ) {
  // set pointer to the node that should follow the new one
  ListNode<T> * p_next_node = itr.p_current_node;

  // set pointer to the node that should precede the new one
  ListNode<T> * p_previous_node = itr.p_current_node->previous;

  // create new node and set its contents
  ListNode<T> * p_new_node = new ListNode<T>;
  p_new_node->element = new_element;
  p_new_node->next = p_next_node;
  p_new_node->previous = p_previous_node;

  // set next and previous nodes to point to the new node
  p_previous_node->next = p_new_node;
  p_next_node->previous = p_new_node;

  return iterator( p_new_node );
}

Figure 13.11: The insert method
**template <class T>**

**inline ListIterator<T> List<T>::erase( iterator itr )** {

  // set pointer to the node to be deleted
  ListNode<T> * p_target_node = itr.p_current_node;

  // set pointer to the node that follows the target node
  ListNode<T> * p_next_node = p_target_node->next;

  // set pointer to the node that precedes the target node
  ListNode<T> * p_previous_node = p_target_node->previous;

  // modify the list to skip the target node
  p_previous_node->next = p_next_node;
  p_next_node->previous = p_previous_node;

  // deallocate the target node
  p_target_node->next = NULL;
  p_target_node->previous = NULL;
  **delete** p_target_node;

  return iterator( p_next_node );

}

Figure 13.12: The **erase** method
Exercises

13.3.2. Add the equality testing operator (==) to ListIterator.

13.3.3. Add the equality testing operator (==) to List.

13.3.4. Add constant iterators to List. Do this by creating a class ConstListIterator that is just like ListIterator except that its dereferencing operator returns a reference to a constant element. Add the type const_iterator to List. Create constant versions of the begin and end functions that return constant iterators.

13.3.5. Add the methods remove(e) and erase(start, stop) to List. Implement them by using erase(itr).

13.3.6. Redo the implementation of erase(start, stop) you did for the previous exercise but this time implement the method directly, without using erase(itr). Do this carefully so this implementation can be more efficient than the one that uses erase(itr).

13.3.7. Add reverse. Make sure your implementation works for lists of odd and even length. Implement the function without creating or destroying any nodes.

13.4 Destroying and Copying Linked Lists

Our List objects point to dynamically allocated nodes. And each List objects “owns” its own nodes. Therefore, as was the case with our implementation of vectors, we need to add a destructor, copy constructor and assignment operator to List.

A List destructor is shown in Figure 13.13. The destructor begins by deleting all the elements of the list by using the pop_back operation. This is done in a loop that runs while the list is not empty. We can tell when the list becomes empty by comparing the begin and end iterators. After all the elements of the list have been deleted, the dummy head node is deallocated. Note that it would be more efficient to use the size or empty methods. An exercise asks you to do this.

A possible implementation of a List copy constructor is shown in Figure 13.14. After creating a dummy head node and initializing the list to be
template <class T>
List<T>::~List() {
  // erase all elements
  while ( begin() != end() ) pop_back();

  // deallocate dummy head node
  delete pHeadNode;
}

Figure 13.13: The destructor

empty, we simply go through the argument and add a copy of each of its
elements to the back of the receiver.

A List assignment operator is shown in Figure 13.15. The operator first
creates a copy of the argument rhs. This means that the actual copying
of the elements (and nodes) is done by the copy constructor. We then ex-
change the nodes of the copy with those of the receiver by using the generic
algorithm swap. The old contents of the receiver, which is now the contents
of copy_of rhs, will be discarded by the destructor when the assignment
operator returns.

The complete declaration of the class List is shown in Figure 13.16. It
shows all the methods we implemented in this chapter. The entire source code
is available on the course web site under List1.2.

Exercises

13.4.1. Revise the destructor so it uses the size or empty methods you im-
plemented for an exercise earlier in this chapter.

13.4.2. Implement the destructor without using other methods such as erase.
   If done carefully, a more direct implementation can be more efficient.
   Hint: After deallocating the first node, there is no need to link the
   second node to the dummy head node.

13.4.3. Implement the copy constructor without using other methods such as
   push_back. If done carefully, a more direct implementation can be more
   efficient. Hint: There is no need to link the last node with the head node
   until the very end.
template <class T>
List<T>::List( const List<T> & ls ) {
    // initialize the receiver to be empty
    p_head_node = new ListNode<T>;
    p_head_node->next = p_head_node;
    p_head_node->previous = p_head_node;

    // traverse argument and add elements at end of receiver
    for ( const ListNode<T> * p_node = ls.p_head_node-&gt;next;
             p_node != ls.p_head_node;
             p_node = p_node-&gt;next ) {
        push_back( p_node-&gt;element );
    }
}

Figure 13.14: The copy constructor

template <class T>
inline List<T> & List<T>::operator=( const List<T> & rhs ) {
    List<T> copy_of_rhs = rhs;
    std::swap(p_head_node, copy_of_rhs.p_head_node);
    return *this;
    // old contents of receiver is deallocated when copy_of_rhs
    // is destroyed
}

Figure 13.15: The assignment operator
template <class T>
class List
// T is the type of element stored in the List.
{
    public:
        typedef ListIterator<T> iterator;

        List();
        List( const List<T> & ls );
        ~List();

        iterator begin() { return iterator( pHeadNode->next ); }  
        iterator end() { return iterator( pHeadNode ); }  

        T & back() { return pHeadNode->previous->element; }  
        const T & back() const {
            return pHeadNode->previous->element; 
        }

        void push_back( const T & newElement );
        void pop_back();

        iterator insert( iterator itr, const T & newElement );
        iterator erase( iterator itr );

        List<T> & operator=( const List<T> & ls );

    private:
        ListNode<T> * pHeadNode;  // points to the dummy head node  
                                  // of a circular, doubly–linked list
};

Figure 13.16: The complete class declaration of List
Chapter 14

Recursion

In this chapter, we will learn about recursion, a technique that greatly simplifies the design and implementation of many algorithms, including the fast sorting algorithms we will learn later in these notes.

14.1 The Technique

Recursion is a technique for designing algorithms. We will see examples of the usefulness of recursion when we study sorting algorithms later in these notes. For now, however, we introduce recursion using simple examples where recursion is neither needed nor a particularly good idea. These examples are only meant to illustrate the technique.

Consider the problem of printing a line containing $n$ copies of a given character $c$. An algorithm for this problem can be designed very simply by putting the statement `cout << c` in a loop that executes $n$ times. A possible implementation is shown in Figure 14.1.

```c
void print(int n, char c) {
    for (int i = 0; i < n; ++i) {
        cout << c;
    }
    cout << endl;
}
```

Figure 14.1: A simple iterative algorithm
void print(int n, char c) {
    if ( n > 0 ) {
        cout << c;
        print2( n-1, c );
    } else {
        cout << endl;
    }
}

Figure 14.2: A recursive algorithm

An alternative algorithm can be designed as follows. First, print one c. Then ask, what is left to do? The answer is, to print a line containing \( n - 1 \) copies of c. And here is the central idea of recursion: this subtask can be performed by using the algorithm that is being designed as if it was already available:

\[
\text{cout} \ll c; \\
\text{print}( n-1, c );
\]

The function call \text{print}( n-1, c ) is \textit{recursive} because it occurs in \text{print} itself. This recursive call does not create a trivial infinite loop because the function is not being called with the same arguments.

However, as is, this recursive algorithm won’t work: it will just keep on calling itself. What we need is a \textit{base case}, a case where recursion is not used. We also need to make sure that the base case will eventually be reached. The algorithm in Figure 14.2 achieves both these objectives. The base case is when \( n \leq 0 \). In that case, we print an empty line.

In general, the correctness of a recursive algorithm can be established by verifying that it satisfies the following three properties:

1. The algorithm has at least one base case, one where the problem is solved directly, without a recursive call.
2. Every recursive call gets closer to a base case, in such a way that a base case will eventually be reached.
3. The algorithm works when you assume that the recursive calls work.
The first two properties guarantee that the algorithm will eventually terminate. The third property, on the other hand, guarantees that the algorithm does what it is supposed to do.

This last property is a little mysterious. In the case of `print`, it means the following:

\[
\text{print}( n, c ) \text{ correctly prints a line containing } n \text{ copies of } c \\
\text{when you assume that print}( n-1, c ) \text{ correctly prints a line containing } n-1 \text{ copies of } c.
\]

In our case, this statement is true. But why does it guarantee that `print( n, c )` works for every possible value \(n\)?

Consider what the property says for \(n, n-1, \ldots, 1\):

\[
\text{print}( n, c ) \text{ works if print}( n-1, c ) \text{ works} \\
\text{print}( n-1, c ) \text{ works if print}( n-2, c ) \text{ works} \\
\text{print}( n-2, c ) \text{ works if print}( n-3, c ) \text{ works} \\
\vdots \\
\text{print}( 2, c ) \text{ works if print}( 1, c ) \text{ works} \\
\text{print}( 1, c ) \text{ works if print}( 0, c ) \text{ works}
\]

Now, go through these statements in reverse order. We know that `print( 0, c )` works because the base case of the algorithm correctly prints a line containing 0 copies of \(c\). This implies that `print( 1, c )` works. Continuing in this way, we get the following:

\[
\text{print}( 1, c ) \text{ works because print}( 0, c ) \text{ works} \\
\text{print}( 2, c ) \text{ works because print}( 1, c ) \text{ works} \\
\vdots \\
\text{print}( n-2, c ) \text{ works because print}( n-3, c ) \text{ works} \\
\text{print}( n-1, c ) \text{ works because print}( n-2, c ) \text{ works} \\
\text{print}( n, c ) \text{ works because print}( n-1, c ) \text{ works}
\]

Therefore, `print( n, c )` works.

This type of argument can be expressed more formally by using the Principle of Mathematical Induction. At Clarkson, this proof technique is covered in a course like MA211 Foundations.

It is important to realize that each recursive call executes independently from the others. In particular, each recursive call has its own arguments and
Figure 14.3: A sample run of the `print` algorithm
if the array is not empty
    display the first element of the array
    display the rest of the array (recursively)
else
    do nothing

Figure 14.4: Recursive algorithm that displays the contents of an array

\[
\text{template <class T>}
\]
\[
\text{void display( const T a[], int start, int stop )}
\]
\[
\text{// Displays the elements of a in the range (start,stop).}
\]
\[
\text{// Elements are separated by one blank space.}
\]
\[
\text{//}
\]
\[
\text{// PRECONDITION: The indices are valid indices in a.}
\]
\[
\text{//}
\]
\[
\text{// ASSUMPTION ON TEMPLATE ARGUMENT: Values of type T can be}
\]
\[
\text{// displayed using the \text{\ll} operator.}
\]
\[
\{
    \text{if (start < stop) }
    \{
        \text{cout \ll a[start] \ll \text{\textquoteleft } \text{\textquoteleft ;}
        \text{display(a, start+1, stop);}
    \}
    \text{// if start >= stop, do nothing}
\}
\]

Figure 14.5: Implementation of the recursive display algorithm

its own set of local variables. For example, the execution of \text{print( 3, \textquoteleft *\textquoteleft )}
can be illustrated as in Figure 14.3.

We end this section with some additional examples of recursive algorithms. The first one displays the contents of an array. It is shown in Figure 14.4, in pseudocode, and implemented in Figure 14.5. The second algorithm computes the sum of the elements in an array of numbers. It is shown in Figures 14.6 and 14.7. The third algorithm displays the contents of an array in reverse. It is shown in Figures 14.8 and 14.9.

Our last example is the binary search algorithm. Earlier in these notes, we described this algorithm as a loop (see Figure 8.8). But the algorithm can also be described recursively, as shown in Figure 14.10. The idea is that after
CHAPTER 14. RECURSION

if the array is not empty
    compute the sum of all the elements except the first one
    (recursively)
    add the first element to that sum
    return the sum
else
    return 0

Figure 14.6: Recursive algorithm that adds the elements of an array of numbers

template <class T>
T sum( const T a[], int start, int stop )
    // Adds the elements of a in the range [start,stop).
    // The sum is returned.
    //
    // PRECONDITION: The indices are valid indices in a.
    //
    // ASSUMPTION ON TEMPLATE ARGUMENT: Values of type T can be
    // added using the + operator and 0 can be converted to a
    // value of type T.
{
    if (start < stop)
        return a[start] + sum(a, start+1, stop);
    else // start >= stop
        return 0;
}

Figure 14.7: Implementation of the recursive sum algorithm

if the array if not empty
    separate the first element of the array
    display all the elements in reverse, except the first one
    (recursively)
    display the first element of the array
else
    do nothing

Figure 14.8: Recursive algorithm that displays the contents of an array in reverse
template <class T>
void display_reverse( const T a[], int start, int stop )
// Displays, in reverse order, the elements of a in the
// range [start,stop). Elements are separated by one
// blank space.
/
// PRECONDITION: The indices are valid indices in a.
/
// ASSUMPTION ON TEMPLATE ARGUMENT: Values of type T can be
// displayed using the << operator.
{
    if (start < stop) {
        display_reverse(a, start+1, stop);
        cout << a[start] << ' ';
    }
    // if start >= stop, do nothing
}

Figure 14.9: Implementation of the recursive reverse display algorithm

having compared e to the middle of s, what is left to do is search one of the
halves of s. That problem can be solved recursively. An implementation of
the recursive binary search for arrays is shown in Figure 14.11.

Source code for the examples of this section is available on the course web
site under Recursion.

Study Questions
14.1.1. What are the three properties of a correct recursive algorithm?

Exercises
14.1.2. Verify that the recursive algorithms shown in Figures 14.4, 14.6, 14.8
and 14.10 satisfy the three properties of a correct recursive algorithm.

14.1.3. Write a recursive function that computes the number of occurrences of
a given element in an array. The function takes as arguments the array,
a start index, a stop index and an element. The function returns the
number of times the element occurs in the range [start,stop).
Input: a sorted sequence s, an element e

if ( s contains more than one element ) {
    locate middle of s
    if ( e < middle element of s )
        search left half of s
    else
        search right half of s
} else
    compare x to only element in s

Figure 14.10: Recursive version of the binary search algorithm

14.1.4. Write a recursive function that finds the maximum element in a nonempty array. The function takes as arguments the array, a start index and a stop index. The function returns the maximum value that occurs in the range [start, stop).

14.1.5. Modify the function of the previous exercise so that it returns the index of the first occurrence of the maximum value.

14.1.6. Write a recursive function that takes as argument an integer \( n \) and prints the numbers 1, 2, 3, \ldots, \( n \). The function should do nothing if \( n < 1 \).

14.1.7. Repeat the previous question but this time print the numbers in decreasing order.

14.1.8. Repeat again, this time printing \( n, n-1, \ldots, 3, 2, 1 \), \( 3, 2, 1, \ldots, n-1, n \). Write a single function.

14.2 When to Use Recursion

First, why use recursion? The main advantage of recursive algorithms is that they can be simpler than non-recursive algorithms that solve the same problem. This means that recursive algorithms can be easier to find and design, as well as easier to understand, implement and modify. We will soon study efficient
14.2. WHEN TO USE RECURSION

```
template <class T>
int binary_search( const T a[], int start, int stop, const T & e )
// Performs a binary search in a for e. Returns the index of
// e in the range [start,stop). Returns -1 if e is not
// found that range.
//
// PRECONDITION: The indices are valid and the elements in the
// range [start,stop) are sorted in increasing order.
//
// ASSUMPTION ON TEMPLATE ARGUMENT: Values of type T can be
// compared by using the < operator.
{
    int middle;
    int n = stop - start; // number of elements

    if ( n >= 2 ) {
        middle = ( start + stop ) / 2;
        if ( e < a[middle] )
            return binary_search( a, start, middle, e );
        else
            return binary_search( a, middle, stop, e );
    } else if ( n == 1 ) {
        if ( e == a[start] ) return start;
        else return -1;
    } else {
        return -1; // n <= 0
    }
}
```

Figure 14.11: An implementation of the recursive binary search for arrays
sorting algorithms and these will be good examples where recursion makes the algorithms simpler and easier to design.

However, it is not always a good idea to use recursion. The main disadvantage of recursive algorithms is that they can generate a lot of function calls. Function calls take more time than most other operations. But for most recursive functions, the additional time taken by the recursive calls is not very significant. What is usually more important is that a recursive function always uses an amount of memory space at least proportional to the number of recursive calls. This should be clear from the sample run shown in Figure 14.3.

In general, the memory requirements of recursive functions lead to the following guidelines:

1. Try to avoid recursion if the number of recursive calls can be large.

2. Don’t use recursion if the number of recursive calls can be large and there is a simple loop that can solve the problem.

What “large” means depends on the context and the size of the input. But, typically, anything at least linear in the input size is considered large while anything logarithmic in the input size is considered small.

To summarize, you don’t want to use recursion if you already have a simple, efficient non-recursive algorithm that solves the problem. You want to use recursion to design an algorithm when you suspect, or hope, that it will be easier that way and that you may get a simpler algorithm. But then, once you have designed the recursive algorithm, you need to check that it doesn’t use much more memory than necessary.

In light of these comments, the first four recursive algorithms of the previous section (print, display, sum, display_reverse) are actually examples where recursion should not be used because we have simple loops that can solve these problems using only a constant amount of memory. The case of binary search is not as clear-cut. Some would argue that the recursive version is more natural and that the extra logarithmic space shouldn’t be a problem, even for very large input sizes.

Note that there are programming languages in which recursion is the normal mechanism for creating repetition because those languages don’t have general-purpose loops. Examples of such languages are Scheme and Prolog. At Clarkson, these languages are typically studied in the course CS341 Programming Languages.
14.3. **TAIL RECURSION**

**Study Questions**

14.2.1. What is the main advantage and the main disadvantage of recursive algorithms?

**Exercises**

14.2.2. Which of the algorithms you wrote for the exercises of the previous section should have not been designed recursively?

14.3 **Tail Recursion**

We know that the minimum amount of memory used by a recursive function is at least proportional to the number of recursive calls it makes. If that number is large, the algorithm will use a lot of space. In such cases, we probably want to look for a non-recursive algorithm.

Sometimes it is fairly easy to directly transform a recursive algorithm into a non-recursive one. A recursive function is said to be **tail recursive** if every time it runs, at most one recursive call is made and that call is the very last action that the function takes. Among the recursive functions we have seen as examples, `print`, `display` and `binarySearch` are tail recursive while `sum` and `displayReverse` are not.

A tail recursive function can be transformed into a loop by following these three general steps:

1. Turn the recursive case of the function into the body of a loop that executes until the base case is reached.
2. Replace the recursive call by statements that update the arguments of the function.
3. Place the base case so it is executed after the loop terminates.

For example, applying these steps to the recursive `print` function produces the iterative version shown in Figure 14.12. Note that this loop uses a constant amount of memory while the recursive version of `print` uses an amount that’s linear in $n$.

Some compilers are able to make tail recursive functions execute efficiently, as if they had been transformed into loops. In general, whenever a function
calls another one at the very end of its execution, any memory space used by the calling function can be deallocated immediately because the calling function has nothing left to do. Some compilers are able to compile these “last calls” in this way. This is called last-call optimization.

Study Questions

14.3.1. What is a tail recursive algorithm?

14.3.2. How can a tail recursive algorithm be transformed into a loop?

Exercises

14.3.3. Transform the recursive display and binary_search functions into loops by using the above three steps.

14.3.4. Among the functions you wrote for the exercises of the first section of this chapter, which ones are tail recursive? Transform them into loops by using the above three steps.
Chapter 15

Sorting

Two of the most frequent operations performed on a collection of data are to search the collection for a particular element and to sort the data by arranging the elements in some order. And these two operations are related: as we have already seen, sorted data can be searched much more quickly by using algorithms such as the binary search. In this chapter, we will learn sorting algorithms, including two very efficient ones: mergesort and quicksort.

15.1 Selection Sort

Consider the problem of rearranging the elements of an array so that they are in increasing order. This is called sorting. Sorting data is a frequent and very useful operation. As we saw in the last section, sorted data can be searched more efficiently. We may also need to sort data for other purposes. In the following sections, we will learn and analyze four different sorting algorithms. The first two are simple but inefficient. The last two are more complicated but much more efficient. They are also good examples of recursive algorithms.

Our first sorting algorithm is called selection sort. The idea is simple: find the largest element of the array and move it to the last position. Then repeat for the rest of the array. The algorithm is shown in Figure 15.1. A sample run of the algorithm is illustrated in Figure 15.2. The first line shows the initial contents of the array. The second line shows the result of the swap and the subarray that will be recursively sorted. The third line shows the result of sorting that subarray. The fourth line shows the final contents of the array.

Figure 15.3 illustrates the same run of selection sort but this time, the
CHAPTER 15. SORTING

input: an array a and two indices start and stop

if ( (start, stop) contains more than one element ) {
    i_max = index of maximum element in (start, stop)
    swap a[i_max] and a[stop-1]
    sort (start, stop-1)
}

Figure 15.1: The selection sort algorithm

\[
\begin{align*}
[60 & \quad 12 & \quad 37 & \quad 42 & \quad 25 & \quad 38 & \quad 16] \\
[16 & \quad 12 & \quad 37 & \quad 42 & \quad 25 & \quad 38] & \quad 60 \\
[12 & \quad 16 & \quad 25 & \quad 37 & \quad 38 & \quad 42] & \quad 60 \\
[12 & \quad 16 & \quad 25 & \quad 37 & \quad 38 & \quad 42 & \quad 60]
\end{align*}
\]

Figure 15.2: A run of the selection sort algorithm (top level of recursion)

entire recursion is shown, not just the top level. The first half of the lines shows the contents of the array after each swap, before the recursive call. The second half shows these arrays at the end of the recursive calls. The portion of the array being sorted by the current recursive call is shown between the two brackets.

Since selection sort is tail recursive we can easily turn it into a more efficient loop, as shown in Figure 15.4. An implementation of selection sort for arrays is shown in Figure 15.5. Note the use of the generic algorithms swap and max_element.

The analysis of selection sort is simple. Consider its recursive version. Let \( T(n) \) be the time required for sorting an array of size \( n \). We know that to find the maximum element in an array of size \( n \) takes time linear in \( n \). Therefore,

\[
T(n) = T(n-1) + cn \quad \text{(when } n \geq 2) \\
T(1) = a
\]

where \( a \) is the (constant) running time of the base case and \( cn \) is the time required in the recursive case for everything besides the recursive call.

These two equations together are called a recurrence relation because the first equation expresses the value of \( T(n) \) in terms of the value of \( T \) on a smaller argument. Recurrence relations are therefore similar to recursive
15.1. SELECTION SORT

input: an array a and two indices start and stop

while ( [start,stop) contains more than one element ) {
   i_max = index of maximum element in [start,stop)
   swap a[i_max] and a[stop-1]
   ---stop
}

Figure 15.3: A run of the selection sort algorithm (entire recursion)

Figure 15.4: An iterative version of the selection sort algorithm
template <class T>
void selection_sort( T a[], int start, int stop )
// Sorts elements in a in increasing order using the
// selection sort algorithm. Sorts elements in the range
// [start,stop). Sorts according to the < operator.
//
// PRECONDITION: The indices are valid and start occurs
// before stop.
//
// ASSUMPTION ON TEMPLATE ARGUMENT: Values of type T can
// be compared using the < operator.
{
    while ( stop - start > 1 ) {
        swap( *max_element(a + start, a + stop), a[stop - 1] );
        --stop;
    }
}

Figure 15.5: An implementation of selection sort for arrays

algorithms and, in fact, they come up naturally in the analysis of recursive
algorithms.

What we need to do now is extract from the recurrence relation an equation
that no longer expresses $T$ in terms of $T$. This is called solving the recurrence
relation.

This particular recurrence relation is very easy to solve. First, note that
the recurrence relation implies the following set of equations:

\[
T(n) = T(n-1) + cn \\
T(n-1) = T(n-2) + c(n-1) \\
\vdots \\
T(2) = T(1) + c2 \\
T(1) = a
\]

Second, add of all these equations to get

\[
T(n) = a + c(2 + \cdots + n) \\
= a + c\left( \frac{n(n+1)}{2} - 1 \right)
\]
15.2. INSERTION SORT

input: an array a and two indices start and stop

if ( [start, stop) contains more than one element ) {
    sort [start, stop-1)
    insert a[stop-1] into [start, stop-1)
}

Figure 15.6: The insertion sort algorithm

This implies that $T(n) = \Theta(n^2)$.

Source code for the implementations of selection sort shown in this section is available on the course web site under Sorting.

Exercises

15.1.1. Run the selection sort algorithm on an array containing the following elements:

12 37 25 60 16 42 38

Show the top level of the recursion, as in Figure 15.2. Then show the contents of the array at the beginning and end of every recursive call, as in Figure 15.3.

15.2 Insertion Sort

In the recursive version of the selection sort algorithm, the recursive sorting of the subarray is done at the end. What if we tried to do it at the beginning? After the recursive call, we would only have to move the last element of the array to its correct position. This gives us a sorting algorithm called insertion sort, which is shown in Figure 15.6. Note that inserting $a[\text{stop-1}]$ into $[\text{start}, \text{stop-1})$ causes one of the elements of the subarray to overflow onto index stop-1.

Figure 15.7 illustrates a run of insertion sort. The first line shows the initial contents of the array. The second line shows the subarray that will be recursively sorted. The third line shows the result of sorting that subarray. The fourth line shows the final contents of the array, after the insertion of the last element into the sorted subarray.
Figure 15.7: A run of the insertion sort algorithm (top level of recursion)

Figure 15.8: A run of the insertion sort algorithm (entire recursion)

Figure 15.8 illustrates the same run of insertion sort but this time, the entire recursion is shown, not just the top level. The first half of the lines shows the contents of the array at the beginning of every recursive call. The second half shows these arrays at the end of the recursive calls. The portion of the array being sorted by the current recursive call is shown between the two brackets.

The analysis of the running time of insertion sort is similar to but just a little more complicated than that of selection sort. Once again, let $T(n)$ be the time required for sorting an array of size $n$. Besides the recursive call, the algorithm needs to insert an element $e$ into a sorted subarray of size $n - 1$. One way to do that is to scan the subarray from right to left looking for the correct location of $e$. While we scan, we shift all the elements one position to the right. When we finally find the correct location of $e$, there is room for it
in the subarray and we just copy it there.

The time required for the insertion depends on the location of \( e \) in the sorted subarray. In the best case, \( e \) is larger than all the elements of the array and no scanning and shifting is necessary. The best-case running time is constant. In the worst case, \( e \) is smaller than all the elements of the array and the entire array must be scanned and shifted. The worst-case running time is \( \Theta(k) \), where \( k \) is the size of the array.

Now, is there, for every \( n \), an array that would cause all these insertions to be the worst possible? The answer is yes: an array sorted in reverse. For this particular, all the insertions will take time \( \Theta(k) \).

Let \( T(n) \) be the running time of insertion sort on array of size \( n \) that’s sorted in reverse. This is also the worst-case running time of insertion sort. Then

\[
T(n) = T(n-1) + cn \quad (\text{when } n \geq 2)
\]

\[
T(1) = a
\]

where \( a \) is the running time of the base case and \( cn \) is the time required in the recursive case for everything besides the recursive call.

This recurrence relation is identical to the one for selection sort. So here too we get that \( T(n) = \Theta(n^2) \).

What about the average-case running time? If each element of the original array is chosen at random, then the same will be true for each of the subarrays to be sorted. Therefore, each insertion will be performed on a random sorted array. On average, the running time of these insertions is also \( \Theta(k) \). Therefore, the average-case running time of insertion sort is also \( \Theta(n^2) \).

The insertion sort algorithm is not tail recursive. So we cannot use the standard technique to easily turn the recursion into a loop. But we can design an iterative version of insertion sort by focusing on the second half of the recursion, that is, on the work that is done after we return from the recursive calls. Essentially, this means writing a loop that performs insertions as in the second half of Figure 15.8. The details are left as an exercise.

**Exercises**

15.2.1. Run the insertion sort algorithm on an array containing the following elements:

\[
12 \quad 37 \quad 25 \quad 60 \quad 16 \quad 42 \quad 38
\]
input: an array

if ( array contains more than one element ) {
    sort the first half of array
    sort the second half of array
    merge the two sorted halves
}

Figure 15.9: The mergesort algorithm

Show the top level of the recursion, as in Figure 15.7. Then show the contents of the array at the beginning and end of every recursive call, as in Figure 15.8.

15.2.2. What is the best-case running time of insertion sort? Clearly identify a best-case input.

15.2.3. Implement the recursive version of the insertion sort algorithm.

15.2.4. Implement an iterative version of the insertion sort algorithm.

15.3 Mergesort

It is possible to design a faster sorting algorithm by considering that insertion sort, like a recursive version of the sequential search algorithm, recurses on a subarray that’s only one smaller than the initial array. The binary search algorithm, on the other hand, divides the array in half. This reduces the number of levels in the recursion to $\Theta(\log n)$ and this reduction is the key factor in the speed of the algorithm. If we could design a sorting algorithm that divides the array in half, the depth of the recursion would be $\Theta(\log n)$ and this may allow the algorithm to run in time $\Theta(n \log n)$.

Figure 15.9 presents such an algorithm, the **mergesort** algorithm. After recursively sorting the two halves of the array, the algorithm merges these sorted arrays back into a single sorted array. Note that this is our first example of a recursive algorithm that makes two recursive calls.

Figure 15.10 illustrates a run of mergesort. The first line shows the initial contents of the array. The second line shows the two subarrays that will be recursively sorted. The third line shows the result of sorting these subarrays.
15.3. **MERGESORT**

The fourth line shows the final contents of the array, after the merging of the two sorted subarrays.

Figure 15.11 illustrates that same run of mergesort but it shows the entire recursion, not just the top level. The first half of the lines show the initial contents of the array at each level of the recursion, as if all the recursive calls at that level were executed simultaneously. The second half of Figure 15.11 shows, for each level of the recursion, the final contents of the array, after the merge step.

It is possible to implement mergesort non-recursively, but this requires more effort and the resulting algorithm is more complicated and no more efficient than the recursive version. Therefore, mergesort is an excellent example of the usefulness of recursion.

The performance of mergesort relies on an efficient merging algorithm. Two sorted arrays can be merged by repeatedly choosing the smallest among the leading elements of both subarrays. A run of this algorithm is illustrated in Figure 15.12. It is clear that the running time of this algorithm is $\Theta(s)$, where $s$ is the total size of the arrays to be merged. Note that the running time of the merge algorithm is the same for all inputs of size $s$.

An implementation of mergesort for arrays is shown in Figure 15.13. This implementation uses the STL generic algorithm `inplace_merge`. 
First array    Second array    Resulting array

[12 37 42 60]   [16 25 38]     []
[37 42 60]      [16 25 38]    [12]
[37 42 60]      [25 38]      [12 16]

[37 42 60]      [38]          [12 16 25]
[42 60]         [38]          [12 16 25 37]
[42 60]         []             [12 16 25 37 38]
[]              []             [12 16 25 37 38 42 60]

Figure 15.12: A run of the merging algorithm

```cpp
template <class T>
void mergesort( T a[], int start, int stop )
// Sorts elements in a in increasing order using the
// mergesort algorithm. Sorts elements in the range
// [start,stop). Sorts according to the < operator.
//
// PRECONDITION: The indices are valid and start occurs
// before stop.
//
// ASSUMPTION ON TEMPLATE ARGUMENT: Values of type T can
// be compared using the < operator.
{
    if ( stop - start > 1 ) {
        int middle = (start + stop) / 2;

        mergesort(a, start, middle);
        mergesort(a, middle, stop);

        inplace_merge(a+start, a+middle, a+stop);
    }
}
```

Figure 15.13: An implementation of mergesort for arrays
15.3. MERGESORT

We now analyze the running time of mergesort. Let $T(n)$ be the running time of the algorithm on arrays of size $n$. As we did for binary search, to simplify the calculations, we are going to assume that $n$ is a power of 2. The general case can be dealt with just like we did for binary search.

When sorting an array of size $n = 2^k$, mergesort sorts two arrays of size $2^{k-1}$ and then merges them. As we mentioned before, the merging can be done in linear time. Therefore,

\[
T(2^k) = 2T(2^{k-1}) + c2^k \quad \text{(when } k \geq 1) \\
T(1) = a
\]

As before, we can consider the following series of equations:

\[
T(2^k) = 2T(2^{k-1}) + c2^k \\
T(2^{k-1}) = 2T(2^{k-2}) + c2^{k-1} \\
\vdots \\
T(2) = 2T(1) + c2 \\
T(1) = a
\]

But if we add these we won’t get all the cancellations we got before because the $T$ values on the right have a constant factor of 2.

To solve this problem, we can multiply each equation by an appropriate constant:

\[
2T(2^k) = 2T(2^{k-1}) + c2^k \\
2T(2^{k-1}) = 2^2T(2^{k-2}) + c2^k \\
2^2T(2^{k-2}) = 2^3T(2^{k-3}) + c2^k \\
\vdots \\
2^{k-1}T(2) = 2^kT(1) + c2^k \\
2^kT(1) = a2^k
\]

Now, if we add all of these, we get

\[
T(2^k) = kc2^k + a2^k
\]

Since $n = 2^k$, we have that $k = \log n$, so that $T(n) = cn \log n + an = \Theta(n \log n)$. 


Note that what each equation
\[ 2^iT(2^{k-i}) = 2^iT(2^{k-i-1}) + c2^k \]
represents, for \( i = 0, \ldots, k \), is the total running time of all the recursive calls at level \( i + 1 \) in the recursion.

Also note that mergesort takes exactly the same amount of time on every array of size \( n \). Therefore, the running time of mergesort is \( \Theta(n \log n) \) in the worst case and on average.

Source code for the implementation of mergesort shown in this section, as well as an implementation of the merging algorithm for arrays, is available on the course web site under Sorting.

**Exercises**

15.3.1. Run mergesort on an array containing the following elements:

\[ 22 \ 37 \ 25 \ 60 \ 16 \ 42 \ 38 \ 46 \ 19 \]

Show the top level of the recursion, as in Figure 15.10. Then show the entire recursion, as in Figure 15.11.

15.3.2. Run the merging algorithm on the following two sorted arrays:

<table>
<thead>
<tr>
<th>First array</th>
<th>Second array</th>
</tr>
</thead>
<tbody>
<tr>
<td>[16 22 25 37 60]</td>
<td>[19 38 42 46]</td>
</tr>
</tbody>
</table>

Illustrate this run as in Figure 15.12.

**15.4 Quicksort**

Any sorting algorithm must examine every element in the input array so the running time of any sorting algorithm must be at least \( \Theta(n) \). Note that this statement applies to every sorting algorithm not just one of them. It is a statement about the computational complexity of the sorting problem itself, not about the running time of a particular algorithm.

Is it possible to sort in time \( \Theta(n) \)? Anything smaller than \( \Theta(n \log n) \)? If the elements to be sorted are numbers, then there are algorithms such as counting sort that can sort in linear time (in exchange for possibly using a lot of memory).
But a generic sorting algorithm should make as little assumptions as pos-
sible about the elements to be sorted. One possibility is to require only that
elements be comparable (with the $<$ operator, for example). In that case, it is
possible to show that the running time of mergesort is the best possible: ev-
ery comparison-based algorithm has a running time that’s at least $\Theta(n \log n)$.
Even on average. (At Clarkson, this result is usually proven in a course such
as CS344 *Algorithms and Data Structures*.)

However, there is a sorting algorithm that in practice tends to run faster
than mergesort. Of course, it can only run faster by a constant factor. We
have studied three sorting algorithms so far: selection sort, insertion sort and
mergesort. All three algorithms work by dividing the array in two parts, recur-
sively sorting one or two subarrays and combining these sorted subarrays. In
the case of selection sort and insertion sort, the array is divided very unevenly
into a single element on one side and all other elements on the other side. This
leads to quadratic running times. Mergesort achieves $\Theta(n \log n)$ by dividing
the array as evenly as possible.

It is interesting to note, however, that insertion sort and mergesort have
something in common: both algorithms divide the array quickly and then
spend most of the effort on combining the two sorted subarrays. (In the case of
insertion sort, one of these subarrays is just a single element.) Selection sort, on
the other hand, spends most of the effort on dividing the array and essentially
no effort on combining the resulting sorted subarray with the element that
was set aside. Can we design an algorithm that divides the array as evenly as
possible, like mergesort, but that spends most of the effort into dividing the
array, like selection sort, so that essentially no effort is required to combine
the two sorted halves?

A key observation is that combining two sorted halves would be trivial if
all the elements in one half were smaller than all the elements in the other
half. This observation leads to the *quicksort* algorithm, which is shown in
Figure 15.14. First, an array element is chosen to play the role of a *pivot*.
Then, the array is divided according to that pivot: all elements smaller than
the pivot to the left, all the others to the right. After recursively sorting the
two subarrays, there is nothing left to do.

Like mergesort, quicksort is an example of a recursive algorithm that makes
two recursive calls. And as with mergesort, it is also possible to implement
quicksort non-recursively, but this requires more effort and the resulting al-
gorithm is more complicated and no more efficient than the recursive version.
Therefore, quicksort is another good example of recursion.
input: an array

if ( array contains more than one element ) {
    choose a pivot element
    partition the array around the pivot
    sort each subarray
}

Figure 15.14: The quicksort algorithm

\[
\begin{array}{cccccccc}
60 & 12 & 37 & 42 & 25 & 38 & 16 \\
12 & 37 & 25 & 38 & 16 & 42 & [60] \\
12 & 16 & 25 & 37 & 38 & 42 & [60] \\
12 & 16 & 25 & 37 & 38 & 42 & 60 \\
\end{array}
\]

Figure 15.15: A run of the quicksort algorithm (top level of recursion)

Figure 15.15 illustrates a run of quicksort. The first line shows the initial contents of the array. The second line shows the two subarrays that will be recursively sorted. Note that 42 was chosen as pivot. The third line shows the result of sorting these subarrays. The fourth line shows the final contents of the array.

Figure 15.16 illustrates that same run of quicksort but by showing the entire recursion, not just the top level. The first half of the lines show the initial contents of the array at every level of the recursion. As if all the recursive calls at a certain level were executed simultaneously. The elements that were chosen as pivots are 42, 16 and 37. The second half of Figure 15.16 shows, for every level of the recursion, the final contents of the array, after the recursive sorting of the subarrays.

As you may have realized from the above illustrations, the performance of quicksort depends heavily on the choice of pivot. We want a pivot that splits the array as evenly as possible but also a pivot that doesn’t take too long to find. In terms of splitting, the ideal pivot is the median element of the array. The median can be found in linear time but the algorithm is not that simple, with the consequence that the hidden constant factors are somewhat large. This would lead to a \( \Theta(n \log n) \) algorithm, but not one that’s likely to be faster than mergesort.
A much faster option is to simply pick the first element of the array as pivot. Unfortunately, this can lead to the worst possible sequence of partitions in the case where the array is already sorted. And, in practice, it is not that uncommon for at least large portions of the input array to be already sorted.

We can avoid this by choosing the pivot to be a random element of the array. This gives a bad partition only if the pivot is near the smallest or largest element of the array. The probability that this should happen very often is extremely small. So a random pivot almost always gives good partitions but it requires calling a pseudo-random number generator, which takes longer than simply choosing the first element of the array.

A good compromise is to chose the median of the first, middle and last elements of the array. It is possible to construct arrays that would be unevenly partitioned under this choice, but it's hard to imagine that such arrays would occur frequently in practice. The runs illustrated in Figures 15.15 and 15.16 were obtained by using this choice of pivot.

We now analyze the running time of quicksort, under the assumption that the pivot is chosen according to one of the last three options we just described. Let $T(n)$ be the running time of the algorithm on arrays of size $n$. When sorting an array of size $n$, quicksort chooses a pivot, partitions the array and then recursively sorts the two parts. The pivot is chosen in constant-time. The partitioning takes linear time. Therefore,

\[
T(n) = cn + T(n_1) + T(n_2) \quad \text{ (when } n \geq 2) \\
T(1) = a
\]

where $n_1$ and $n_2$ are the sizes of the two parts.

The running time depends in a critical way on how the array is split. The best possible split is $n_1 = n_2 = n/2$. If this always occurred, at every level of
the recursion, then this recurrence relation would be identical to the one we
encountered in the analysis of mergesort, and the running time of quicksort
would be \( \Theta(n \log n) \).

A bad split would be \( n_1 = n - 1 \) and \( n_2 = 0 \). Because of the way we are
choosing the pivot, there is at least one array that will cause all the splits to
be bad. In that case, quicksort essentially degenerates into selection sort and
the running time is quadratic.

So the worst-case running time of quicksort is at least quadratic. To show
that it’s no larger than that, let \( T^{(i)}(n) \) be the total running time of all the
recursive calls running at level \( i \) of the recursion. At the top level of the
recursion, level 0, only one call is running, the original one. Therefore, \( T(n) =
T^{(0)}(n) \).

Now, at each level of the recursion, a certain number of arrays are parti-
tioned. We don’t know how many but we know that their total size is at most
\( n \). Which implies that the partitions take time no greater than \( cn \) in total. In
addition, a certain number of calls running at each level of the recursion are
base cases. There can’t be more than \( n \) of those so their total running time is
at most \( an \). Therefore,

\[
T^{(i)}(n) \leq cn + an + T^{(i+1)}(n)
\]

How many levels are there in this recursion? At each level, the subarrays
are at least one smaller than at the previous level. Therefore, all the recursive
calls made at level \( i \) are on arrays of size at most \( n - i \). So we know that by the
time we get to level \( n - 1 \), all the recursive calls that are running are on arrays
of size at most 1, which means that these calls are all base cases. Therefore,

\[
T^{(n-1)}(n) \leq an
\]

These two equations give us a recurrence relation that we can solve as
before. Let \( b = c + a \). Then,

\[
\begin{align*}
T^{(0)}(n) & \leq bn + T^{(1)}(n) \\
T^{(1)}(n) & \leq bn + T^{(2)}(n) \\
\vdots \\
T^{(n-2)}(n) & \leq bn + T^{(n-1)}(n) \\
T^{(n-1)}(n) & \leq an
\end{align*}
\]
Adding all these equations gives $T^{(0)}(n) \leq bn(n - 1) + an$. This shows that the worst-case running time of quicksort is at most quadratic and, therefore, exactly quadratic.

So the worst-case running time of quicksort is quadratic, just like insertion sort and selection sort. However, quicksort does tend to run much faster in practice, even slightly faster than mergesort, which has a $\Theta(n \log n)$ running time. This may be explained by the fact that the average-case running time of quicksort is $\Theta(n \log n)$. At Clarkson, this result is usually proven in CS344 Algorithms and Data Structures.

**Study Questions**

15.4.1. What is the fastest possible asymptotic running time of a comparison-based sorting algorithm?

15.4.2. What are the worst-case and average-case running times of quicksort?

**Exercises**

15.4.3. Run quicksort on an array containing the following elements:

```
22  37  25  60  16  42  38  46  19
```

Show the top level of the recursion, as in Figure 15.15. Then show the entire recursion, as in Figure 15.16. Use the first element as pivot.

15.4.4. Describe an array that would partitioned as evenly as possible by choosing the median of the first, middle and last elements as pivot. Describe an array that would partitioned in the worst possible way by this choice of pivot.

15.4.5. What is the best-case running time of quicksort? Clearly identify a best-case input for each of the possible choices of pivot discussed in this section.

15.4.6. Implement quicksort.
Appendix A

Library Reference

This appendix is a reference to some of the components of the C++ standard library.

A.1 C Strings

See Section 3.1 for an introduction to C strings. Table A.1 lists several C string operations. All are defined in the library file cstring, except for the I/O functions and operators, which are defined in iostream. The C string functions are in the global namespace (and not in the std namespace like most elements of the C++ standard library.) Additional C string functions are described in an online reference such as cplusplus.com [CPP].
**APPENDIX A. LIBRARY REFERENCE**

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>strlen(cs)</code></td>
<td>Returns the length of C string <code>cs</code>.</td>
</tr>
<tr>
<td><code>strcpy(dest, source)</code></td>
<td>Makes C string <code>dest</code> a copy of C string <code>source</code>. The second version copies at most <code>n</code> characters. If that maximum is reached, the null character is <em>not</em> appended to <code>dest</code>.</td>
</tr>
<tr>
<td><code>strncpy(dest, source, n)</code></td>
<td></td>
</tr>
<tr>
<td><code>strcat(dest, source)</code></td>
<td>Appends a copy of C string <code>source</code> to C string <code>dest</code>. The second version copies at most <code>n</code> characters, followed by a null character.</td>
</tr>
<tr>
<td><code>strncat(dest, source, n)</code></td>
<td></td>
</tr>
<tr>
<td><code>strcmp(cs1, cs2)</code></td>
<td>Returns a negative integer if <code>cs1 &lt; cs2</code>, 0 if <code>cs1 == cs2</code>, a positive integer if <code>cs1 &gt; cs2</code>. Uses alphabetical order. The second version copies at most <code>n</code> characters.</td>
</tr>
<tr>
<td><code>strncmp(cs1, cs2, n)</code></td>
<td></td>
</tr>
<tr>
<td><code>stream &lt;&lt; cs</code></td>
<td>Outputs the characters of C string <code>cs</code>.</td>
</tr>
<tr>
<td><code>stream &gt;&gt; cs</code></td>
<td>Reads characters into C string <code>cs</code>. Skips leading white space and stops reading at white space (blank, tab or newline) or at the end of the file. The terminating character is not read.</td>
</tr>
<tr>
<td><code>stream.get(cs, n)</code></td>
<td>Reads at most <code>n−1</code> chars into C string <code>cs</code>. Does not read past the end of the current line. The null character is appended to <code>cs</code>. Does not read the newline character, if encountered.</td>
</tr>
<tr>
<td><code>stream.getline(cs, n)</code></td>
<td>Reads the rest of the current input line into C string <code>cs</code>. Does not read more than <code>n−1</code> chars. The null character is appended to <code>cs</code>. Reads the newline character, if encountered, but does not add it to <code>cs</code>.</td>
</tr>
</tbody>
</table>

Table A.1: Some C string operations
A.2 C++ Strings

See Section 3.2 for an introduction to C++ strings. The class string is defined in the library file string and included in the std namespace. Tables A.2 to A.8 show most of the string operations. Additional operations are described in an online reference such as cplusplus.com [CPP]. In these tables, the word string without qualifier refers to C++ strings.
string s
string s(s2)
string s(s2, i, n)
string s(a, n)
string s(n, c)

Creates a string $s$ that is initialized to be empty, a copy of string $s2$, a copy of the substring of $s2$ that starts at index $i$ and is of length $n$, a copy of the first $n$ characters of array $a$, or $n$ copies of character $c$. The argument $s2$ can also be a C string.

s.length()
s.size()

Asks string $s$ for the number of characters it currently contains.

s.empty()

Asks string $s$ if it is empty.

s.max.size()

Asks string $s$ for the maximum number of characters it can contain.

s[i]

Returns a reference to the character at index $i$ in string $s$.

s.c.str()

Asks string $s$ for a C string that contains the same characters as $s$.

s.substr(i, m)
s.substr(i)

Asks string $s$ for a copy of the substring that starts at index $i$, and is of length $m$ or ends at the end of the string.

s.copy(a, m)
s.copy(a, m, i)

Asks string $s$ to copy $m$ characters to array a. The copying starts at the beginning of $s$ or at index $i$. The number of characters copied is returned.

Table A.2: string constructors and operations that retrieve length information or characters
stream << s
Outputs the characters of string s. Returns a reference to the stream.

stream >> s
Reads characters into string s. Skips leading white space and stops reading at white space (blank, tab or newline). That terminating character is not read. Returns a reference to the stream.

ggetline(stream, s)
Reads characters into string s until then end of the current line. The newline character is read but not included in s. Returns a reference to the stream.

Table A.3: string I/O operations

s.resize(n)
s.resize(n, c)
Asks string s to change its size to n. If n is smaller than the current size of s, the last characters of s are erased. If n is larger, s is padded with the null character or with copies of character c.

s1 = s2
Makes string s1 a copy of string s2. The right operand can also be a C string or a single character. Returns a reference to s1.

s.assign(s2)
Asks string s to become a copy of string s2. The argument can be of any of the forms accepted by the constructors. A reference to s is returned.

s1.swap(s2)
Asks string s1 to swap contents with string s2.

Table A.4: string operations that resize or assign strings
\begin{tabular}{|l|}
\hline
\texttt{s1 + s2} & Returns a string that consists of a copy of string \texttt{s1} followed by a copy of string \texttt{s2}. One of the operands must be a \texttt{string} object but the other can be a C string or a single character. \\
\texttt{s1 += s2} & Appends a copy of string \texttt{s2} to string \texttt{s1}. The right operand can also be a C string or a single character. A reference to \texttt{s} is returned. \\
\texttt{s.append(s2)} & Asks string \texttt{s} to append to itself a copy of string \texttt{s2}. The argument can be of any of the forms accepted by the constructors. A reference to \texttt{s} is returned. \\
\texttt{s.push_back(c)} & Asks string \texttt{s} to append to itself a copy of character \texttt{c}. \\
\hline
\end{tabular}

Table A.5: string operations that concatenate strings
s.insert(i, s2)
    Asks string s to insert into itself, at index i, a copy of string s2. The second argument can be of any of the forms accepted by the constructors. A reference to s is returned.

s.replace(i, m, s2)
    Asks string s to replace the substring of length m that starts at index i by a copy of string s2. The third argument can be of any of the forms accepted by the constructors. A reference to s is returned.

s.clear()
    Asks string s to delete all its characters.

s.erase(i, m)
s.erase(i)
s.erase()
    Asks string s to delete m characters starting at index i, or all the characters from index i to the end of the string, or all the characters in the string. A reference to s is returned.

Table A.6: string operations that add or remove characters
s1.find(s2)
s1.find(s2, i)
s1.find(a, i, m)
    Asks string s1 for the index of the first occurrence of string s2, or of the string formed by the first m characters of array a. In the first version, the entire string is searched. In the others, the search starts at index i. The argument s2 can also be a C string or a single character. If the search is unsuccessful, −1 is returned.

s1.find first of(...)  
    Similar to find except that the search is for any of the characters contained in the given string, not the entire string.

s1.find first not of(...)  
    Similar to find except that the search is for any character not contained in the given string.

s1.rfind(...)  

s1.find last of(...)  

s1.find last not of(...)  
    Similar to find, find first of and find last of except that the search is for the last occurrence and that the search ends at index i.

Table A.7: string methods that search for characters in strings
A.2. C++ STRINGS

s1 op s2

Comparing string s1 with string s2 where the operator op is one of ==, !=, <, >, <= or >=. Uses alphabetical order. Returns true or false. One of the operands must be a string object but the other can be a C string.

s.compare(s2)
s.compare(i, n, s2)
s.compare(i, n, s2, j, m)
s.compare(i, n, a, m)

Asks string s to compare two strings. In the first version, the first string is all of s. In the other versions, the first string is the substring of s that starts at index i and is of length n. The second string is either string s2, or the substring of s2 that starts at index j and is of length m, or the string formed by the first m characters of array a. The argument s2 can also be a C string. The strings are compared according to alphabetical order. If the first string is greater than the second string, a positive integer is returned. If the first string is less than the second string, a negative integer is returned. If the two strings are equal, 0 is returned.

Table A.8: string operations that compare strings
cout
A buffered output stream (class ostream) normally associated with the computer screen.

A buffered input stream (class istream) normally associated with the keyboard.

Table A.9: Standard input and output streams

A.3 I/O Streams

See Section 3.5 for an introduction to I/O streams. Table A.9 describes the two standard streams cin and cout. Table A.10 lists several istream and ostream operations. In that table, out and in refer to output and input streams, respectively. Table A.11 lists operations that are specific to file streams. Table A.12 lists error-related stream operations. The streams cin and cout, as well as the classes istream and ostream, are defined in the library file iostream. The file stream classes are defined in the library file fstream. All of these classes and objects are part of the std namespace. Additional stream objects and operations provided by the C++ standard library are described in a reference such as cplusplus.com [CPP].
A.3. I/O STREAMS

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>out &lt;&lt; data</strong></td>
<td>Sends data to the output stream and returns the stream.</td>
</tr>
<tr>
<td><strong>out.put(c)</strong></td>
<td>Asks the output stream to write character c. The stream is returned.</td>
</tr>
<tr>
<td><strong>in &gt;&gt; var</strong></td>
<td>Reads from the input stream a value of the appropriate type and stores it in the variable. Initial white space is skipped. Returns the stream.</td>
</tr>
<tr>
<td><strong>in.get()</strong></td>
<td>Asks the input stream to read and return the next character.</td>
</tr>
<tr>
<td><strong>in.get(var)</strong></td>
<td>Asks the input stream to read the next character and store it in the variable. The stream is returned.</td>
</tr>
<tr>
<td><strong>in.peek()</strong></td>
<td>Asks the input stream to return the next character without removing it from the stream. The stream is returned.</td>
</tr>
<tr>
<td><strong>in.putback(c)</strong></td>
<td>Asks the input stream to put back character c so it will be the next character that is read. The stream is returned.</td>
</tr>
</tbody>
</table>

Table A.10: Some input and output operations
<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>type f</code></td>
<td>Creates a file stream <code>f</code> of the specified type (ifstream, ofstream or fstream).</td>
</tr>
<tr>
<td><code>type f(cs_file_name)</code></td>
<td>Creates a file stream <code>f</code> and asks it to open the file with the given name. The argument is a C string.</td>
</tr>
<tr>
<td><code>f.open(cs_file_name)</code></td>
<td>Asks file stream <code>f</code> to open the file with the given name. The argument is a C string.</td>
</tr>
<tr>
<td><code>f.close()</code></td>
<td>Asks file stream <code>f</code> to close the file currently associated with it.</td>
</tr>
</tbody>
</table>

Table A.11: Some operations specific to file streams
stream.eof()
    Asks the stream if it is in the eof state (an attempt was made to
    read past the end of the file).

stream.fail()
    Asks the stream if it is in the fail state (on a previous attempt to
    read, the data was not in the correct format, there was no more data
    in the file, or the stream was not associated with a properly opened
    file).

stream.bad()
    Asks the stream if it is in the bad state (on a previous attempt to
    read, an error other than the above occurred).

stream.good()
    Asks the stream if it is in the good state (that is, not in any of the
    eof, fail or bad states).

stream.clear()
    Asks the stream to return to the good state.

stream.setstate(errorbit)
    Asks the stream to add the error state specified by errorbit.
    The argument is one of eofbit, failbit or badbit. These values
    can be accessed as stream.eofbit, stream.failbit and
    stream.badbit.

Table A.12: Error-related stream operations
Table A.13: Some operations specific to string streams

<table>
<thead>
<tr>
<th>Operation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>type ss()</code></td>
<td>Creates a string stream <code>ss</code> of the specified type (istringstream, ostrstream or stringstream). The second version sets the string associated with the stream to be a copy of string <code>s</code>.</td>
</tr>
<tr>
<td><code>ss.str()</code></td>
<td>Returns a copy of the string associated with string stream <code>ss</code>.</td>
</tr>
<tr>
<td><code>ss.str(s)</code></td>
<td>Sets the string associated with string stream <code>ss</code> to be a copy of string <code>s</code>.</td>
</tr>
</tbody>
</table>

### A.4 String Streams

See Section 3.8 for an introduction to string streams. String streams support all the general stream operations described in Section A.3. Table A.13 lists operations that are particular to string streams. The string stream classes are defined in the library file `sstream`. These classes are part of the `std` namespace. Additional stream operations provided by the C++ standard library are described in a reference such as `cplusplus.com [CPP]`.

### A.5 Vectors

See Chapter 5 for an introduction to vectors. Tables A.14 and A.15 shows some of the most basic vector operations. The `vector` container is defined in library file `vector` and included in the `std` namespace. Additional operations supported by STL vectors are described in a reference such as `cplusplus.com [CPP]`.
A.5. VECTORS

vector<T> v
vector<T> v(n)
vector<T> v(n, e)
vector<T> v(v2)

Creates a vector v that can hold elements of type T. The vector is initialized to be empty, or to contain n copies of the default object of class T, or n copies of element e, or to be a copy of vector v2.

v.size()
Asks vector v for the number of elements it currently contains.

v.empty()
Asks vector v if it is empty.

v.max_size()
Asks vector v for the maximum number of elements it can contain.

v[i]
Returns a reference to the element at index i in vector v.

v.front()
v.back()
Asks vector v for a reference to its front or back element.

Table A.14: vector constructors and operations that retrieve length information or elements
v.resize(n)
v.resize(n, e)
    Asks vector v change its size to n. If n is smaller than the current size of v, the last elements of v are deleted. If n is larger than the current size, then v is padded with either copies of the default object of class T or with copies of element e.

v.push_back(e)
    Asks vector v to add a copy of element e to its back end.

v.pop_back()
    Asks vector v to delete its last element.

v.clear()
    Asks vector v to delete all its elements.

v1 = v2
    Makes vector v1 a copy of vector v2. Returns a reference to v1.

v.assign(n, e)
    Asks vector v to change its contents to n copies of element e.

v1.swap(v2)
    Asks vector v1 to swap contents with vector v2.

Table A.15: Some additional vector operations
Bibliography


Index

abstract data type (ADT), 23
abstraction, 18
data, 23
procedural, 18
activation record, 99
agile software development, 191
argument
  constant reference, 8
default, 40
asymptotic
  analysis, 134
equivalence, 133
automatic variable, 101

balanced binary search tree, 169
binary search, 147
class, 23
client code, 116
cohesion, 16
computational complexity, 268
constructor, 36
  compiler-generated, 39
copy, 119
default, 36
explicit, 115
container, 91
conversion
  implicit, 39
copy
  deep, 89, 120
  shallow, 89, 120
coupling, 17
delete, 103
derived class, 65
design, of software, 4, 187
destructor, 119
diagram
  component, 22
  interaction, 205
dynamic variable, 101
encapsulation, 90
equivalence relation, 142
evolution, of software, 189
exception, 76
extreme programming, 191
fail-safe software, 73
file
  header, 50
  implementation, 50
friend, 24
genetic, 91
  algorithm, 213, 221
class, 112
  function, 110
  programming, 110
hash table, 169
head node
dummy, 228
helper methods, 210
implementation, of software, 9, 189
incremental development, 4, 191
independence, 16
information hiding, 17
inheritance, 67
initializer, 180
inline, 33
input validation, 79
insertion sort, 261
integrated development environment (IDE), 53
iterative development, 4, 191
iterator, 157
  begin, 157
  bidirectional, 220
  category, 219
  constant, 157
  end, 158
  linked list, 233
  random-access, 220
  reverse, 219
Java, 28
last-call optimization, 256
life cycle, of software, 189
linked list, 154
  circular, 229
  traversal, 230
list, 155
maintenance, of software, 189
make, 55
Makefile, 55
member
  data, 24
  function, 27
memory allocation
  automatic, 101
  dynamic, 100
memory leak, 103
mergesort, 264
message, 26
method, 26
  constant, 32
  get/set, 41
mixed-case format, 8
modularity, 16
  advantages, 17
naming convention, 8, 42
new, 100
node, linked list, 227
null
  character, 57
object, 26
  default, 38
operator
  address-of (\&), 210
  decrementing (\texttt{--}), 158
  dereference-and-select (\texttt{\rightarrow}), 172, 230
  dereferencing (*), 123, 158, 172
  incrementing (\texttt{++}), 158
  overloading, 44
pointer, 100
  \texttt{NULL}, 103
this, 123
polymorphism, 67
preconditions, 73
private, 24
programming
  imperative, 26
object-oriented (OOP), 24, 69
public, 27
quicksort, 269
range, 158
receiver, 26
recurrence relation, 258
recursion, 245
reliable software, 73
responsibility-driven design, 196
reuse, of software, 225
robust software, 73
running time
  asymptotic, 133
  average-case, 144
  worst-case, 144
scenarios, 192
searching
  sequential search, 222
selection sort, 257
specification, of software, 1, 187
standard software components
  benefits, 222
Standard Template Library (STL), 91
stream
  I/O, 65
  string, 69
string
  C, 57
  C++, 61
subclass, 65
tail recursion, 255
template
  class, 112
  compiling, 115
  function, 109