# Introduction to Data Abstraction, Algorithms and Data Structures

*With C++ and the STL*

Spring 2016

Alexis Maciel
Department of Computer Science
Clarkson University

# Contents

# 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, vectors (or 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. 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, a large program always contains a large amount of details and all this information can be difficult to manage. The solution is to organize the program 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, a large program typically holds a large amount of data that needs to be accessed efficiently. In response, computer scientists have invented a wide variety of *data structures*, which are ways of organizing data so it can be stored and accessed efficiently. You are already familiar with one data structure: vectors (or arrays). In these notes, you will learn about other basic data structures: 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 vectors and linked lists.

Third, a large program often performs complex tasks that require nontrivial techniques. Computer scientists have designed *algorithms* that perform a wide variety of complex 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 software development process, 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 are the foundation you need 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.

These notes were typeset using LaTeX (MiKTeX implementation with the TeXworks environment). The paper size and margins are set small to make it easier to read the notes on a small screen. If the notes are to be printed, it is recommended that they be printed double-sided and at "Actual size", not resized to "Fit" the paper. Otherwise, you'll get pages with huge text and tiny margins.

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 to 1.3 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 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?

```
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 program begins by asking the user for the names of
the input and output files, as follows:

    Name of input file: times.txt
    Name of output file: report.txt

After reading the name of the input file, the program
attempts to open it. In case of failure, the message

    Could not open file.

is printed and the program halts.

After reading the name of the output file, the program
attempts to open it. In case of failure, the message

    Could not open output file.

is printed and the program halts.
```

Figure 1.1: Specification of the pay calculator (part 1 of 3)

```
The input file should contain one line for each
employee.  Each line should contain 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 program writes to the output file 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.
```

Figure 1.2: Specification of the pay calculator (part 2 of 3)

```
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 other error—checking is performed in this version
of the program.
```

Figure 1.3: Specification of the pay calculator (part 3 of 3)

## 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 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 1.4 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, be-

```
Components of the program:

  o A function that runs the calculator

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

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

  o 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 1.4: First draft of the design of the pay calculator

```
o const double kPayRate = 12;


o int main()

  Runs the calculator.  See program spec for details.

  Implementation note: Uses Time to store times and
  delegates the time operations to the associated
  functions.
```

Figure 1.5: Design of the pay calculator (part 1 of 2)

tween 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 to more modularity.

Note that it is not clear at this point if we need an operation that initialize times. This depends on exactly how the function that runs the calculator will be coded. For example, if we read times immediately after they are declared, then it is not really necessary to initialize the times and it is more efficient not to. Since an initialization operation may be needed and is often useful, we are including it in the design of our program.

Figures 1.5 and 1.6 show the final design of the pay calculator. It includes a global constant for the pay rate. This makes the program easier to modify. This global constant can be considered a minor component of the program.

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

```
o 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:

      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.  In other words, the
          difference is computed as "t1 — t2".
```

Figure 1.6: Design of the pay calculator (part 2 of 2)

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 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 `find_entry`. In addition, when the main purpose of a function is to return a value, the name of the function will be a noun phrase that describes that value, as in `difference`. On the other hand, when the main purpose of a function is to perform an action, the name of the function will be a verb phrase that describes the action, as in `read` and `print`.

One exception to these rules concerns *compile-time constants*. These are constants whose value is determined before the program runs. The pay calculator contains one of these constants: `kPayRate`. 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.

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.

In the next section, the `main` function of the pay calculator will use a stream variable to read from the input file. This variable could be called `input_file`. But this would not make it 100% clear that the variable is a stream and not just the name of the input file. For this reason, we will use the name `ifs_input`. The prefix `ifs` will make it clear that this variable is an input file stream.

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 fol-

lowed. 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 `ofs` in the name `ofs_output`?

## 1.4 Implementation

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

Figure 1.7 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 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 1.8. 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're bound to make at least some 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

```
struct Time
{
    int hours;
    int 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 1.7: The `Time` data type and its associated functions

```
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 1.8: A test driver for `Time` and its operations

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.

Figures 1.9 and 1.10 show the implementation of the `main` function of the pay calculator.

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.

```
int main()
{
    cout << "Name of input file: ";
    string input_file_name;
    getline(cin, input_file_name);

    ifstream ifs_input(input_file_name);
    if (!ifs_input) {
        cout << "Could not open file.\n";
        return 1;
    }

    cout << "Name of output file: ";
    string output_file_name;
    getline(cin, output_file_name);

    ofstream ofs_output(output_file_name);
    if (!ofs_output) {
        cout << "Could not open output file.\n";
        return 1;
    }

    ...
}
```

Figure 1.9: The `main` function of the pay calculator (part 1 of 2)

```
int main()
{
    ...

    int employee_number;
    while (ifs_input >> employee_number) {
        Time start_time;
        read(start_time, ifs_input);

        Time stop_time;
        read(stop_time, ifs_input);

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

        ofs_output << employee_number << ' ';
        print(start_time, ofs_output);
        ofs_output << ' ';
        print(stop_time, ofs_output);
        ofs_output << " $" << fixed << setprecision(2)
                   << pay << '\n';
    }

    return 0;
}
```

Figure 1.10: The `main` function of the pay calculator (part 2 of 2)

1.4.7. Create a data type `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 `initialize(date)` and

```
initialize(date, month, day, year)
```

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

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

c) An operation `print(date, out)` that prints `date` to output stream `out`. Dates are printed in numerical format, as in `1/22/2012`.

d) An operation `print_in_words(date, out)` that also prints `date` to output stream `out` but with the month in words, as in `January 22, 2012`.

e) An operation `add(date, num_days)` that advances `date` by the number of days `num_days`. The argument `num_days` is an integer. That number could be arbitrarily large and even negative. *Hint*: This is where the assumption that every month has exactly 30 days is useful.

1.4.8. Create a data type `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 `initialize(v)` and `initialize(v, x, y, z)` that initialize `ThreeDVector v` to `(0, 0, 0)` and `(x, y, z)`, respectively.

b) An operation `read(v, in)` that reads `ThreeDVector v` 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) An operation `print(v, out)` that prints `ThreeDVector v` to output stream `out`. Vectors are printed in the format described for `read`.

d) An operation `add(v1, v2)` that returns the sum of `ThreeDVector`'s `v1` and `v2`.

1.4.9. Create a data type `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

```
initialize(f)
initialize(f, a)
initialize(f, a, b)
```

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 `read(f, in)` that reads `Fraction f` 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) An operation `print(f, out)` that prints `Fraction f` to output stream `out`. Fractions are printed in the format described for `read`. Fractions are not reduced.

d) An operation `print_mixed(f, out)` that prints `Fraction f` 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 which the numerator is less than the denominator). For example, −`5 6/8`. The fraction is not reduced.

e) Operations `add(r, s)` and `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 `add(r, s)` returns a `Fraction` whose value is 17/12 and `multiply(r, s)` returns 6/12. The returned fractions are not reduced.

1.4.10. 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 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.*, as in `8:35 a.m.` or `12:10 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. Each line in these files contains an employee number and a time. In both files, employees are listed in increasing order of employee number.

d) The program reads another file called `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 wage 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 time file (or files). (But you can assume that every employee that appears in the time files also appears in the wage file.) *Hint*: Start with a first version where you assume that every employee works every day.

## 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 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.

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 1.11: Advantages of modularity

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 1.11 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 `print` of the pay calculator. We know *what* that function does: it reads a time. We also know *how* that function does what it does: those implementation details are shown in Figure 1.7. What is critical to note is that a function that uses `print`, such as `main`, depends on *what* `print` does, not on *how* it does it.

For example, in the implementation of `print`, there are other ways in which we could ensure that the minutes are always printed with two digits. If we decided to use another method, the implementation of `print` would need to be revised but the other functions of the program would not need to change because they can only depend on what `print` does, not on how it does it.

Independence between software components 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 **abstraction**. In the case of functions, we call it **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 *abstract* component. This is the *outside* or *public* view of the component. This view concerns only *what* the component does and it includes the *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 *inside* or *private* view. It concerns *how* the component does what it does and it includes the *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 `main` depends on what the function `print` does but not on how it does it. As a consequence, if the implementation of `print` was modified, there would be no need to modify `main`.

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 1.7). 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 1.7 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 `main` function?

By examining the implementation of `main` (Figures 1.9 and 1.10), we can see that there is nothing in this code that depends on the fact that each `Time` value is a structure consisting of two integers `hours` and `minutes`. In particular, `main` 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. Note that the arrow going from `main` to `Time` does not go *inside* the box that represents the data type. This reflects the fact that `main` 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 `main` 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 `main` 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

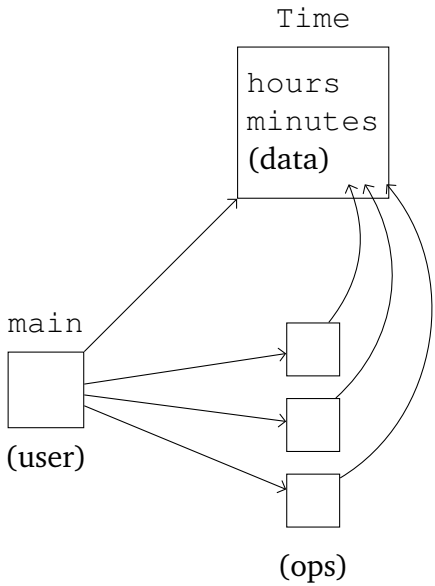


Figure 2.1: Component diagram of the pay calculator

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 `main` works with `Time` values by using the `Time` operations instead of accessing the `Time` data directly. In other words, `main` treats the `Time` data type as an ADT. But nothing prevents `main` 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 `main` 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.

```
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;
    int minutes;
};
```

Figure 2.2: A `Time` class with friend operations

Note that we didn't have to turn `Time` into a class; we could have left it as a structure. In fact, the only difference between structures and classes in C++ is that, by default, data is private in a class and public in a structure (meaning that it can be accessed without restrictions). In these notes, we will follow the common practice of using structures only when all the data is public.

Some programming languages don't have provide any notion of privacy. This can be because they're older languages, such as C, or because they are meant to be simpler, such as JavaScript. In those languages, data abstraction cannot be enforced. We must instead rely on programmer discipline.

In this section, we used classes and privacy to enforce data abstraction. But classes allow us to do more than that, 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.

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 read `start_time` by calling the appropriate function:

```
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 read itself:

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

On receiving the `read` message, `start_time` responds by executing the corresponding method (which would be defined in the class `Time`). Note how `read` illustrates the fact that messages 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:

```
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:

```
stop_time.minus(start_time)
```

Note that in this last example, the two `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 `main` function that uses the new class `Time`. (Only the portion of `main` that needed to be modified is shown. Compare with Figure 1.10.) 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

```
class Time
{
public:
    void initialize() { hours = minutes = 99; }

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

    void print(ostream & out)
    {
        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;
    int minutes;
};
```

Figure 2.3: A class `Time` with methods

```
int main()
{
    ...

    int employee_number;
    while (ifs_input >> employee_number) {
        Time start_time;
        start_time.read(ifs_input);

        Time stop_time;
        stop_time.read(ifs_input);

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

        ofs_output << employee_number << ' ';
        start_time.print(ofs_output);
        ofs_output << ' ';
        stop_time.print(ofs_output);
        ofs_output << " $" << fixed << setprecision(2)
                   << pay << '\n';
    }

    return 0;
}
```

Figure 2.4: A revised `main` function

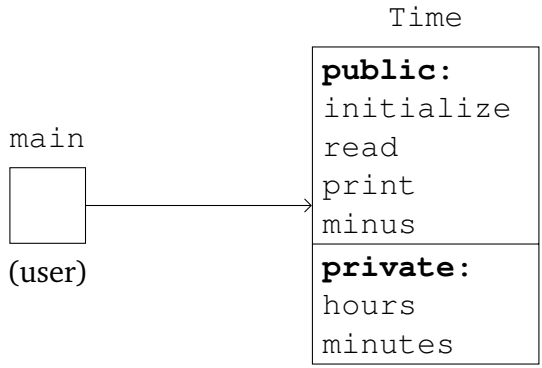


Figure 2.5: An object-oriented version of the pay calculator

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 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 1.4.7 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.

2.3.7. Turn the data type `ThreeDVector` of Exercise 1.4.8 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 1.4.9 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

```
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 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:

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

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

```
void println(const Time & t, ostream & out)
{
    t.print(out);
    out << '\n';
}
```

Figure 2.6: The `println` function

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`, the method `minus` should also be declared constant:

```
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 may remove the function call and replace 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 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 [Goo].

The methods of our class `Time` are all short and simple enough that it makes sense to make them inline. But what if we decided that a method should not be inline? How could we achieve this? We can make a method not be inline by declaring the method inside the class declaration but implementing it outside. For example, Figure 2.7 shows a version of `Time` in which the method `print` is no longer inline. Note that in the implementation of the method, its name is preceded by the name of the class: `Time::print`. This tells the compiler that `print` is a method that belongs to the class `Time` and not a *standalone function* (a function that is not a method of any class).

Inline methods can make a class declaration become crowded. The style preferred by many C++ programmers is for a class declaration to be mainly a list

```
class Time
{
public:
    void print(ostream & out) const;
...
};

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

Figure 2.7: A non-inline version of the `print` method

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 line are now implemented outside the class declaration. Note that these methods are still inline because their implementations are preceded by the keyword **`inline`**.

So methods can be made inline by implementing them within the class declaration, or by implementing them outside and using the keyword **`inline`**. In the case of standalone functions, the **`inline`** keyword is the only option, as shown in Figure 2.9.

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?

```
class Time
{
public:
    void initialize() { hours = minutes = 99; }
    void read(istream & in);
    void print(ostream & out) const;
    double minus(const Time & t2) const;

private:
    int hours, minutes;
};

inline void Time::read(istream & in)
{
    ...
}

inline void Time::print(ostream & out) const
{
    ...
}

inline double Time::minus(const Time & t2) const
{
    ...
}
```

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

```
inline void println(const Time & t, ostream & out)
{
    t.print(out);
    out << '\n';
}
```

Figure 2.9: An inline standalone 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 line. But make sure short methods stay inline. Use the rule of thumb given in this section.

## 2.6 Constructors

A common programming mistake is to create a variable but 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 `count` is needed for a loop, it is better to set it to its initial value right away, as in

```
int count = 0;
```

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

```
int employee_number;
while (ifs_input >> employee_number) {
    Time start_time;
    start_time.read(ifs_input);

    Time stop_time;
    stop_time.read(ifs_input);

    ...
}
```

Figure 2.10: A portion of `main`

Another example comes from the pay calculator. As shown in Figure 2.10, the variables `employee_number`, `start_time` and `stop_time` are currently not initialized when they are declared because their values are immediately set on the following line. This is what happens when we follow the practice of declaring variables just before their first use.

But another approach is to declare variables at the beginning of the block of code where they will be used. In our case, this would mean declaring `start_time` and `stop_time` at the beginning of the loop. In addition, since `employee_number` must be declared outside of the loop, we could declare those three variables together, just before the loop. This is reasonable because those three variables are closely related: they correspond to the data that must be read from each line of the input file. The result is shown in Figure 2.11.

But now, those variables are not set immediately after they are declared. So it is safer to initialize them as shown in Figure 2.12.

Note that it would be more convenient and even safer if variables were initialized automatically as soon as they are created. In C++, this is possible in the case of objects.

Every time an object is created, it is always automatically initialized by a

```
int employee_number;
Time start_time;
Time stop_time;

while (ifs_input >> employee_number) {
    start_time.read(ifs_input);
    stop_time.read(ifs_input);

    ...
}
```

Figure 2.11: The variables `start_time` and `stop_time` declared before the loop

```
int employee_number = -1;
Time start_time;
start_time.initialize();
Time stop_time;
stop_time.initialize();

while (ifs_input >> employee_number) {
    start_time.read(ifs_input);
    stop_time.read(ifs_input);

    ...
}
```

Figure 2.12: Initialization of `employee_number`, `start_time` and `stop_time`

```
class Time
{
public:
    Time() : hours(99), 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;

private:
    int hours;
    int minutes;
};
```

Figure 2.13: The class `Time` with three constructors

special method called a **constructor**. For example, Figure 2.13 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

```
Time start_time;
```

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 default constructor initializes the `hours` and `minutes` of the time by using two **initializers**:

```
Time() : hours(99), minutes(99) {}
```

An alternative would have been to use assignment statements in the body of the constructor:

```
Time()
{
    hours = 99;
    minutes = 99;
}
```

In the case of data members with a primitive type, such as **`int`**, **`double`**, **`char`** and **`bool`**, this makes no difference. But if the class had included an object as a data member, a `string`, for example, then that object would have been initialized by its default constructor before being reassigned in the body of the `Time` constructor. The initializer version of the `Time` constructor is more efficient because its initializers override the automatic initialization of the data members. In other words, using initializers in constructors ensures that data members are initialized only once.

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;
```

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 is called an **implicit conversion** because it was not explicitly requested by the programmer.

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

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

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

or

```
Time noon = 12;
```

This equal sign is not the assignment operator but just another way of initializing an object.

If you do not include any constructor in a class, then the compiler will automatically generate a default constructor. This compiler-generated default constructor 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 implementations of the three `Time` constructors have most of their code in common. It is possible to eliminate this repetition by having the first two constructors *delegate* their work to the third one as follows:

```
Time() : Time(99, 99) {}
Time(int h) : Time(h, 0) {}
Time(int h, int m) : hours(h), minutes(m) {}
```

We then say that the first two constructors are **delegating constructors**.

We can simplify the constructors a bit more by using **in-class initializers**. The idea is to provide an initial value with the declaration of the data members:

```
int hours = 99;
int minutes = 99;
```

The constructors can then be implemented as follows:

```
Time() {}
Time(int h) : Time(h, 0) {}
Time(int h, int m) : hours(h), minutes(m) {}
```

Because of the in-class initializers, the default constructor has nothing left to do. The second constructor delegates to the third one, as before, while the third constructor uses initializers to override the in-class initializers and set the data members to other values.

In-class initializers are particularly useful when a class has a large number of constructors that set a data member to the same value. This is not the case with our class `Time`.

In fact, it is debatable whether in-class initializers are a good idea in the case of `Time`. One advantage of in-class initializers is that they follow more closely the principle that variables should be initialized as soon as they are declared. But a disadvantage is that the initialization code is split between two different locations: the data member declarations and the constructors. This may make the class harder to understand.

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?

2.6.7. What is an initializer?

2.6.8. What is an advantage of using initializers?

2.6.9. What is a delegating constructor?

2.6.10. What is an advantage of using delegating constructors?

2.6.11. What is an in-class initializer?

2.6.12. What is an advantage and a disadvantage of using in-class initializers?

### Exercises

2.6.13. Revise the classes of the exercises of Section 2.3 by turning their `initialize` methods into constructors. Use delegating constructors and in-class initializers as appropriate.

## 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 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 methods** in the class. For example, Figure 2.14 shows a version of `Time` with two get methods called `hours` and `minutes`. Then, without modifying the class, these methods allow us to write a `print_with_h` function that prints times with an *h* instead of a colon, as shown in in Figure 2.15.

These methods are called get methods because they allow us to retrieve values associated with their receivers. In fact, names such as `get_hours` and `get_minutes` are common alternatives to the simpler `hours` and `minutes`. In these notes, as explained in Section 1.3, we will normally use noun phrases as names for methods whose main purpose is to return a value.

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.

Now, suppose that `t` is a `Time` and that we need to change its minutes to `m`. With the class of the previous section, we could do this as follows:

```
t = Time(t.hours(), m);
```

But the class of Figure 2.14 includes a `set_minutes` method that allows us to do this more conveniently and more efficiently:

```
t.set_minutes(m);
```

A similar `set_hours` method is also included. Not surprisingly, methods such as `set_minutes` and `set_hours` are called **set methods**.

We might also need to change both the hours and minutes of `t` to, say, `h` and `m`. With the class of the previous section, this could be done as follows:

```
t = Time(h, m);
```

```
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;
    }

    void set(int new_hours, int new_minutes = 0);

private:
    int hours_;
    int minutes_;
};

inline void Time::set(int new_hours, int new_minutes)
{
    hours_ = new_hours;
    minutes_ = new_minutes;
}
```

Figure 2.14: Get and set functions for the class `Time`

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

Figure 2.15: The function `print_with_h`

We could now use the set methods:

```
t.set_hours(m);
t.set_minutes(h);
```

But even more convenient and efficient is to use the `set` method included in the class of Figure 2.14:

```
t.set(h, m);
```

Note that the declaration of the `set` method specifies that the value 0 is to be used in case the second argument is missing. This default value is called a *default argument*. This allows us to set `t` to `h:00` by simply doing

```
t.set(h);
```

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.

Note that a default argument could be used to combine the second and third constructors of our class:

```
Time(int h, int m = 0) : hours_(h), minutes_(m) {}
```

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 simplest way to print a time is to use the `print` method as in

```
t.print(out);
```

But now that `Time` has get and set methods, nothing prevents us from printing times ourselves:

```
out << t.hours() << ':';
if (t.minutes() < 10) out << '0';
out << t.minutes();
```

This code is obviously less readable and less convenient to write. But it is also much more dependent on the details of `Time`. For example, if the colon was changed to an `h` or if seconds were added to times, every occurrence of this code would need to be revised. None of this work is needed if we always use the `print` method.

Therefore, as a general rule, it is much better for users of a class not to use 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?

2.7.2. What is a default argument?

## Exercises

2.7.3. 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.

2.7.4. 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 the get and set methods we introduced in this section.)

2.7.5. Add to the class `Date` of Exercise 2.3.6 methods `set(month, day, year)` and `set(month, day)` that set the date to the given month, day and year. The second method leaves the year unchanged. The arguments are integers. Use default arguments as appropriate.

2.7.6. 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.

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

a) Methods `set(a, b)` and `set(a)` that set the fraction to $a/b$ and $a$, respectively. The arguments are integers. Assume that `b` is positive. Use default arguments as appropriate.

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:

```
stop_time.minus(start_time)
```

But the following code is more natural:

```
stop_time - start_time
```

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

Figure 2.16: A subtraction operator for `Time`

And because it is more natural, it 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. This is 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 2.16. When the compiler sees an expression such as

```
stop_time − start_time
```

it will interpret it as

```
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`.

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.

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

Figure 2.17: The subtraction operator as a standalone function

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 stand-alone function with two arguments, as shown in Figure 2.17. This is possible because an expression such as

```
stop_time − start_time
```

can also be interpreted by the compiler as

```
operator−(stop_time, 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

```
start_time.read(ifs_report)
```

we can write

```
ifs_report >> 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 >> t
```

Just as in the case of the subtraction operator, there are two possibilities:

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 standard input stream `cin` as well as all input file streams.[1] 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 overloading the operator by creating a standalone function with two arguments, as shown in Figure 2.18.

Note that here we decided to make the operator a friend of the class so we can directly access its private data members. The alternative would have been to use the set methods as shown in Figure 2.19. This would have been less efficient because it requires storing each number twice.

Note that the input operator returns the stream it receives as argument. There are two reasons for this.

[1] Strictly 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.

```
inline istream & operator>>(istream & in, Time & t)
{
    in >> t.hours_;
    in.get(); // colon
    in >> t.minutes_;
    return in;
}
```

Figure 2.18: An input operator for `Time`

```
inline istream & operator>>(istream & in, Time & t)
{
    int new_hours;
    in >> new_hours;
    t.set_hours(new_hours);

    in.get(); // colon

    int new_minutes;
    in >> new_minutes;
    t.set_minutes(new_minutes);

    return in;
}
```

Figure 2.19: A less efficient input operator

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`:

```
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

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

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

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

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

```
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

```
inline ostream & operator<<(ostream & out,
                            const Time & t)
{
    out << t.hours() << ':';
    if (t.minutes() < 10) out << '0';
    out << t.minutes();
    return out;
}
```

Figure 2.20: An output operator for `Time`

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.

The output (or stream insertion) operator `<<` can be overloaded in a way similar to the input operator, as shown in Figure 2.20.

Figure 2.21 shows a revision of the `main` function of the pay calculator 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 web site 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?

```
int main()
{
    ...

    int employee_number = -1;
    Time start_time, stop_time;

    while (ifs_input >> employee_number) {
        ifs_input >> start_time >> stop_time;

        double pay = (stop_time - start_time) *
                         kPayRate;

        ofs_output << employee_number << ' '
                   << start_time << ' ' << stop_time
                   << " $" << fixed << setprecision(2)
                   << pay << '\n';
    }

    return 0;
}
```

Figure 2.21: A revised `main` function that uses constructors and operators

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`.

2.8.7. Add the operators <<, >> and += to the class `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 `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 * to the class `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 `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 *header file* and an *implementation file*. The header file contains the declarations associated with the component, that is, all the code that the compiler needs in order to compile the code that uses the component. This normally consists of global constants, class declarations, function declarations, as well as the implementation of all inline methods and functions.

The implementation file contains the rest of the code. This normally consists of the implementation of the non-inline methods and functions. The names of these files normally end with the extensions `h` and `cpp`, respectively.

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

1. `Time.h`: the declaration of the class `Time` and the implementation of all the inline `Time` operations (whether methods or standalone functions), as shown in Figure 2.22.

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

```
#ifndef _Time_h_
#define _Time_h_

#include <iostream>

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

    ... // declaration of the methods

private:
    int hours_;
    int minutes_;
};

... // implementation of the inline operations

#endif
```

Figure 2.22: The header file `Time.h`

3. `main.cpp`: the global constant `kPayRate` and 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, `Time.cpp` includes `Time.h` while `main.cpp` includes several library files as well as `Time.h`.

Each header file should include all the necessary library and header files. For example, `Time.h` includes `iostream` (see Figure 2.22).

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 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 `main.cpp` contains declarations such as **`using`** `std::ifstream` and **`using`** `std::cout`.

With a compiler that is part of an *integrated development environment* (IDE),[2] 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
```

[2]Examples of IDE's are *Code::Blocks* and *Eclipse* for Windows, Linux and Mac OS X, *Bloodshed Dev-C++* and *Microsoft Visual C++* for Windows, and *Anjuta* for Linux. As of January 2012, Code::Blocks, Eclipse, Dev-C++, Anjuta and the Express Edition of Visual C++ were all available for free.

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 is used by both the `main` function and the class `Time`. The header file of that new component would be included in both `main.cpp` and `Time.h`. But this would cause problems because `Time.h` is already included in `main.cpp`: when `main.cpp` is compiled, the compiler will read the header file of the new component twice and will complain that the new component is declared more than once.

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 Figure 2.22. At the beginning of this header file, 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 a 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:

```
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:

```
g++ –c Time.cpp
g++ *.o –o paycalc
```

This recompiles `Time.cpp` and then relinks all the object files together.

The difficulty with this manual approach is that if we modify a header file, then we need to figure out which implementation 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

```
paycalc: main.o Time.o
        g++ main.o Time.o -o paycalc

main.o: main.cpp Time.h
        g++ -c main.cpp -o main.o

Time.o: Time.cpp Time.h
        g++ -c Time.cpp -o Time.o

clean:
        rm -f *~ *.o paycalc
```

Figure 2.23: A `Makefile` for the pay calculator program

with a declaration of dependencies between the various source files, as shown in Figure 2.23. For example, the entry

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

indicates that the file `main.o` depends on `main.cpp` and `Time.h`, and that whenever these other files are changed, `main.o` should be regenerated by using the command

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

The 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 2.23 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. Later in these notes, 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 or if ever we needed to implement our own class of strings. In addition, in C++ 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. Tables 3.1 and 3.2 lists several of these C string operations. All are defined in the library file `cstring`, except for `atoi` and `atof`, which are defined in `cstdlib`, and 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 library functions are described in a reference such as [CPP].[1]

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 encountered. 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.

[1] Some of these other functions involve concepts that we still have not covered, such as pointers. You can ignore these functions for now.

`strlen(cs)`
: Returns the length of C string `cs`.

`strcpy(dest, source)`
`strncpy(dest, source, n)`
: Makes C string `dest` a copy of C string `source`. The second version copies at most `n` characters. If that maximum is reached before the null character is copied, then the null character is *not* appended to `dest`.

`strcat(dest, source)`
`strncat(dest, source, n)`
: Appends a copy of C string `source` to C string `dest`. The second version copies at most `n` characters, followed by a null character.

`strcmp(cs1, cs2)`
`strncmp(cs1, cs2, n)`
: Returns a negative integer if `cs1 < cs2`, 0 if `cs1 == cs2`, a positive integer if `cs1 > cs2`. Uses alphabetical order. The second version compares at most `n` characters from each string.

`atoi(cs)`
`atof(cs)`
: Converts C string `cs` into a number. Starting from the beginning of the string, skips whitespace and then converts as many characters as possible into a number. If no conversion can be performed, 0 is returned. The first version returns an **`int`** while the second version returns a **`double`**. The string is not modified.

Table 3.1: Some C string operations (part 1 of 2)

`stream << cs`
: Outputs the characters of C string `cs`.

`stream >> cs`
: Reads characters into C string `cs`. 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.

`stream.get(cs, n)`
: Reads at most `n-1` chars into C string `cs`. Does not read past the end of the current line. The null character is appended to `cs`. Does not read the newline character, if encountered.

`stream.getline(cs, n)`
: Reads the rest of the current input line into C string `cs`. Does not read more than `n-1` chars. The null character is appended to `cs`. Reads the newline character, if encountered, but does not add it to `cs`.

Table 3.2: Some C string operations (part 2 of 2)

```
int my_strlen(const char cs[])
{
    int i = 0;
    while (cs[i] != '\0') ++i;
    // i is both the index of '\0' and the length of
    // the string
    return i;
}
```

Figure 3.1: An implementation of `strlen`

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 address this problem. If `n` is no greater than the size of the array holding `dest`, then `strncpy` will not go out of bounds. But if ever `source` is of length `n`, then `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, `strlen` can be used to compute the length of a literal string: `strlen("abc")` returns 3. And `strcpy` can be used to copy a literal string to another C string: `strcpy(dest, "abc")`. But note that we can't copy anything to a literal string as in `strcpy("abc", 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 `get` and `getline`?

### Exercises

3.1.4. Create a function `println(cs)` that takes a C string as argument and behaves exactly as the code `cout << cs << endl`. (But don't use this code in implementing the function.)

3.1.5. Create a function `strlwr(cs)` that changes to lowercase all the letters in C string `cs`. (You'll probably want to use the function `tolower` from the standard library `cctype`. This function takes a character as 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. Later in these notes, 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 3.3 to 3.5 show several `string` operations. Many more are described in a reference such as [CPP].[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.

The second constructor provides a way for converting C strings into C++ strings. The reverse conversion can be performed by the `c_str` method.[4,5]

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")`

[2]A typical limit is approximately 4 billion characters.

[3]Some of these operations involve concepts we still have not covered, such as iterators, ranges, exceptions and capacity. You can ignore these operations for now.

[4]Before C++11, the latest version of C++, a common use of this operation was for opening files because the file stream constructor that takes a file name as argument accepted a C string but not a C++ string. If we had a file name that was stored in a `string` variable, it was necessary to first convert it into a C string by using the `c_str` method, as in

```
ifstream ifs(s_file_name.c_str());
```

This may be necessary with compilers that still don't support C++11.

[5]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.

`string s`
`string s(s2)`
`string s(s2, i, n)`
`string s(n, c)`
: Creates a string `s` and initializes it to be empty, or a copy of string `s2`, or a copy of the substring of `s2` that starts at index `i` and is of length `n`, 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`.

`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.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 string operations (part 1 of 4)

`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.

`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.

`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`.

`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.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.

Table 3.4: Some string operations (part 2 of 4)

`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.erase(i, m)`
`s.erase(i)`
: Asks string `s` to delete `m` characters starting at index `i`, or all the characters from index `i` to the end of the string. A reference to `s` is returned.

`s1.find(s2)`
`s1.find(s2, i)`
: Asks string `s1` for the index of the first occurrence of string `s2` as a substring. In the first version, the entire string is searched. In the other, the search starts at index `i`. The argument `s2` can also be a C string or a single character. If the search is unsuccessful, the constant `string::npos` ("not a position") is returned.

`s1.rfind(s2)`
`s1.rfind(s2, i)`
: Similar to `find` except that the search is for the *last* occurrence and that the search *ends* at index `i`.

`s.c_str()`
: Asks string `s` for a C string that contains the same characters as `s`. (That C string may become invalid if a subsequent operation modifies the size of `s`.)

Table 3.5: Some string operations (part 3 of 4)

`stoi(s)`
`stod(s)`
: Converts string `s` into a number. Starting from the beginning of the string, skips whitespace and then converts as many characters as possible into a number. The first version returns an **`int`** while the second version returns a **`double`**. The string is not modified.

`to_string(x)`
: Returns a string representing number `x`. The argument can be of any of the usual numeric types.

`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.

`getline(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 3.6: Some string operations (part 4 of 4)

```
#include <cstdlib>
#include <string>

inline int stoi(const std::string & s)
{
    return std::atoi(s.c_str());
}
```

Figure 3.2: An implementation of `stoi`

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 directly.

The `find` and `rfind` methods return the special value `string::npos` ("not a position") in case the search is unsuccessful. If `s` is a string, that constant can also be accessed as `s.npos`.

The functions `stoi`, `stod` and `to_string` can be used to easily perform conversions between numbers and their string representations. These functions are new to C++11 but note that some compilers that support C++11 do not properly support these functions.[6] In this case, the `stoi` function can be implemented by using the C string `atoi` function as shown in Figure 3.2. The `stod` function can be implemented by using `atof` in the same way. Figure 3.3 shows the easiest implementation of `to_string`. This implementation uses a *string stream*, a type of stream we will cover later in this chapter.

[6] As of February 2014, this was still the case with the version of the g++ compiler that comes with the Code::Blocks IDE.

```
#include <sstream>
#include <string>

inline std::string to_string(int x)
{
    std::ostringstream oss;
    oss << x;
    return oss.str();
}
```

Figure 3.3: An implementation of `to_string`

## 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 function.)

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 the same name but in the format `"Doe, John"`.

a) Write your code by using only the default constructor, the indexing operator and the methods `length` and `resize`.

| | |
|---|---|
| `cout` | A buffered output stream (class `ostream`) normally associated with the computer screen. |
| `cin` | A buffered input stream (class `istream`) normally associated with the keyboard. |

Table 3.7: Standard input and output streams

b) Now simplify your code by using the method `find` and the operator `+=`.

c) Simplify your code further by using the method `append`. (Consult a reference such as [CPP].)

## 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 3.7. These objects are of class `istream` and `ostream`, respectively. These two classes provide several useful operations. Some of these are listed in Table 3.8. 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

`out << data`
: Sends data to the output stream and returns the stream.

`out.put(c)`
: Asks the output stream to write character `c`. The stream is returned.

`in >> var`
: 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.

`in.get()`
: Asks the input stream to read and return the next character.

`in.get(var)`
: Asks the input stream to read the next character and store it in the variable. The stream is returned.

`in.peek()`
: Asks the input stream to return the next character without removing it from the stream. The stream is returned.

`in.putback(c)`
: Asks the input stream to put back character `c` so it will be the next character that is read. The stream is returned.

Table 3.8: Some input and output operations

`type f`
: Creates a file the stream of the specified type (`ifstream`, `ofstream` or `fstream`).

`type f(file_name)`
: Creates a file stream and asks it to open the file with the given name. The argument can be a C++ string or a C string.

`f.open(file_name)`
: Asks the file stream to open the file with the given name. The argument can be a C++ string or a C string.

`f.close()`
: Asks file the stream to close the file currently associated with it.

Table 3.9: Some operations specific to file streams

these two classes are included in `fstream`. Table 3.9 includes some operations that are available only with file streams.

Note that when a file stream variable ceases to exist, then the associated file is automatically closed. This happens, for example, when a stream variable is local to a function and that function returns. In such a case, it is not necessary to explicitly ask the stream to close the file.

Note also 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 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*.[7]

At any moment, a stream is in at least one of the following **states**: end of file, error or good. A stream enters the **end of file** state if an attempt was made to read past the end of the file. Note that this is not necessarily an error since it will occur, for example, when a number is read and that number was the last piece of data in a text file.

A stream generally enters the **error** state if the last stream operation failed. Note that a stream can be in both the end of file and error states, or in one or the other. Note also that if a stream is in the end of file or the error state, then every subsequent input operation on that stream will fail.

A stream is in the **good** state if it is not in the end of file or in the error state. This essentially means that the stream is ready to be read from.

The stream classes provide methods to determine if a stream is in any of these states, as shown in Table 3.10. An additional `clear` method is provided to return the stream to the good state. This method is useful for error recovery.

When a stream is used as a Boolean expression, it evaluates to **`true`** if it is not in the error state. In other words, a stream is true if the last stream operation was successful. We already took advantage of this in our pay calculator, for two different purposes: (1) to check that the files opened successfully (see Figure 1.9) and (2) to control the loop that reads the input file (see Figure 1.10). Note that the usual logical operators, including negation (!), can be used on streams.

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 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 a reference such as [CPP].

[7] 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 end of file state (an attempt was made to read past the end of the file).

`stream.fail()`
: Asks the stream if it is in the error state (an error occurred).

`stream.bad()`
: Asks the stream if a non-recoverable error occurred. (This is a special case of the error state.)

`stream.good()`
: Asks the stream if it is in the good state (and ready to be read from).

`stream.clear()`
: Asks the stream to return to the good state (usually for error recovery).

`stream`
: Evaluates to **`true`** if the stream is not in the error state (an error did not occur).

Table 3.10: Error-related stream operations

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 objects and operations shown in Tables 3.7 to 3.10.

3.3.2. Create functions `read(cs, n, in)` and `readln(cs, n, in)` that behave exactly as the C string functions `get` and `getline` described in Table 3.2. (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.)

`type ss()`
`type ss(s)`
: Creates a string stream `ss` of the specified type (`istringstream`, `ostringstream` or `stringstream`). The second version associates the stream with a copy of string `s`.

`ss.str()`
: Asks string stream `ss` for a copy of its string.

`ss.str(s)`
: Asks string stream `ss` to set its string to be a copy of string `s`.

Table 3.11: Some operations specific to string streams

## 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. 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.

String streams support all the general stream operations described in Section 3.3. That's because the string stream classes `istringstream`, `ostringstream` or `stringstream` are subclasses of the I/O stream classes `istream`, `ostream` and `iostream`. Table 3.4 lists some operations that are particular to string streams.

To illustrate the usefulness of string streams, we will add some additional error checking to the pay calculator program we created in the previous two chapters. Right now, the program does not check for errors in the input file. If that file is incorrectly formatted, the program will fail silently, producing incorrect results without any warning.

Let's change that. We already check that the input and output files open properly. Now we will also check that the employee numbers and start and stop times are read properly. In case of an error, we will report the number of the line on which the error occurred.

The easiest way to report line numbers, is to read each line of the input file into a string variable and then extract the employee number, start time and stop time from that string. This is where string streams come in handy.

Figure 3.4 show a portion of the revised `main` function. The stream manipulator `ws` is used to extract whitespace from the line to make sure that it contains nothing else besides the employee number and the start and stop times.

Note that the pay calculator program does not check for every possible error. In particular, the program does not check that times are properly formatted and correspond to valid times. For example, the times `8:5` and `37:50` would not be recognized as errors. We will address this in the next chapter.

The string stream classes are defined in the library file `sstream`. These classes are part of the `std` namespace. The version of the pay calculator we created in this section is available on the course web site under `PayCalculator2.4`.

## 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.

```
string s_line;
int line_number = 0;

while (getline(ifs_input, s_line)) {
    ++line_number;

    // extract data from line
    istringstream iss_line(s_line);
    int employee_number = -1;
    Time start_time, stop_time;
    iss_line >> employee_number >> start_time
             >> stop_time;

    // quit if an error occurred
    if (!iss_line) {
        cout << "Error in input file at line "
             << line_number << '\n';
        return 1;
    }

    // quit if rest of line is not blank
    iss_line >> ws;
    if (!iss_line.eof()) {
        cout << "Error in input file at line "
             << line_number << '\n';
        return 1;
    }

    ...
}
```

Figure 3.4: Error checking in the pay calculator

# 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

We say that a program is *reliable* when it does what it is supposed to do when used properly. Ideally, we would also like our programs to be *robust* or *fail-safe*; that is, we would like them to always behave in a reasonable way, *no matter what*. Essentially, this means that our programs should test for all 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.

```
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;
    }
}
```

Figure 4.1: Different ways of reporting errors

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

Figure 4.2: The method `is_valid`

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.

## 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 errors 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 any 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 exceptions by revising our pay calculator program once again. The last version of the program, the one we created in Section 3.4, performed some error checking. If an error was discovered, the program printed an error message to standard output and halted.

We are now going to imagine that the pay calculator is part of a larger program. We will replace the `main` function by a function called `run_pay_calculator` that will be called by this other program. Instead of printing error messages, the function `run_pay_calculator` will report errors to its calling function. And it will do that by throwing exceptions.

Figures 4.3 and 4.4 show most of the implementation of the function `run_pay_calculator`. Note how **`throw`** statements are used to throw exceptions.

```
void run_pay_calculator()
{
    cout << "Name of input file: ";
    string input_file_name;
    getline(cin, input_file_name);

    ifstream ifs_input(input_file_name);
    if (!ifs_input)
        throw PayCalculatorError("File " +
            input_file_name + " did not open");

    cout << "Name of output file: ";
    string output_file_name;
    getline(cin, output_file_name);

    ofstream ofs_output(output_file_name);
    if (!ofs_output)
        throw PayCalculatorError("File " +
            output_file_name + " did not open");

    ...
}
```

Figure 4.3: A `run_pay_calculator` function that throws exceptions (part 1 of 2)

```
void run_pay_calculator()
{
    ...

    string s_line;
    int line_number = 0;

    while (getline(ifs_input, s_line)) {
        ++line_number;

        // extract data from line
        int employee_number = -1;
        Time start_time, stop_time;
        istringstream iss_line(s_line);
        iss_line >> employee_number >> start_time
                 >> stop_time;

        // check if an error occurred
        if (!iss_line)
            throw PayCalculatorError(
                "Error in input file at line " +
                to_string(line_number));

        ...
    }
}
```

Figure 4.4: A `run_pay_calculator` function that throws exceptions (part 2 of 2)

```
class PayCalculatorError
{
public:
    PayCalculatorError(const string & d)
        : description_(d) {}
    const string & what() const
    {
        return description_;
    }
private:
    string description_;
};
```

Figure 4.5: The exception class `PayCalculatorError`

In each case, an exception of class `PayCalculatorError` is thrown. Figure 4.5 shows the declaration of this class. 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.

Note that the constructor initializes the description by using an initializer. As discussed in Section 2.6, an alternative would have been to use an assignment statement in the body of the constructor:

```
    PayCalculatorError(const string & d)
    {
        description_ = d;
    }
```

But this would cause the description to be first initialized with the default constructor of class `string` and then assigned the value of `d`. The initializer version causes the description to be immediately set to `d` by the `string` constructor

```
try { run_pay_calculator(); }

catch (const PayCalculatorError & e) {
    cout << "Pay calculator could not complete ("
         << e.what() << ")\n";
}
```

Figure 4.6: Catching exceptions

that takes another string as argument. The initializer version is therefore more efficient.

Figure 4.6 shows how exceptions thrown by the function `run_pay_calculator` can be caught. The function call is placed within a **`try`** statement that's followed by a **`catch`** clause. If `run_pay_calculator` throws an exception, then the code within the **`catch`** clause is executed.

If `run_pay_calculator` is used in a larger program, then the context in which the function is called determines how the exception should be caught. For example, if the code that normally follows the calculation of the pay cannot be executed when the pay calculator fails, then the setup shown in Figure 4.7 would be appropriate. In this case, if `run_pay_calculator` throws an exception, then the execution of the **`try`** statement is immediately halted and the **`catch`** clause is executed.

On the other hand, if it is possible to recover from a failure of the pay calculator, then the setup shown in Figure 4.8 should be used instead. A key difference is that the exception is caught closer to the point where it is thrown.

The main advantage of exceptions, as we mentioned earlier, is that they are essentially impossible to ignore. Another advantage, as shown in Figure 4.7, is that in some situations, exceptions allow us to isolate the error handling code (inside the **`catch`** clause) from the "normal" code (inside the **`try`** statement). The error handling code becomes almost a footnote to the normal code. When

```
try {
    // code that precedes the calculation of the pay
    run_pay_calculator();
    // code that follows the calculation of the pay
}

catch (const PayCalculatorError & e) {
    // error handling code
}
```

Figure 4.7: Catching exceptions when recovery is not possible

```
// code that precedes the calculation of the pay

try { run_pay_calculator(); }
catch (const PayCalculatorError & e) {
    // error handling code
}

// code that follows the calculation of the pay
```

Figure 4.8: Catching exceptions when recovery is possible

error flags are used, this separation is usually not possible since error flags must almost always be tested immediately after the function call that sets or returns them. This causes the error handling code to be intertwined with the normal code, making that code messier.

On the other hand, exceptions also have disadvantages. First, when looking at a function call, there is nothing in that code that tells us if the function may throw an exception, what type of exception it may throw, or where execution will resume in case an exception is thrown. Fully understanding code that uses exceptions can require more work.

Second, if we call a function that may throw an exception and forget to catch the exception (or make the conscious decision not to catch it), the compiler will not say anything. On the other hand, if the function sets an error flag argument and we forget to include it in the call, the compiler will point out the error. And if a function returns an error flag and we forget to examine it, the compiler will often issue a warning.

It is interesting to note that both of these disadvantages are related to the fact that exceptions allow us to separate the normal code and the error handling code.

To minimize the disadvantages of exceptions, it is generally recommended that exceptions be used only for errors that are truly exceptional, errors that are not expected to occur. For example, errors in user input are not exceptional; on the contrary, they can be expected to occur. Error flags and invalid states should probably be used instead of exceptions in this case. On the other hand, if an input file is generated by another function in the same program, then that file is internal to the program and it may be reasonable to assume that it would contain errors only under exceptional circumstances.

There is continuing debate on whether the advantages of exceptions outweigh their disadvantages. Some argue that exceptions are great but too hard to program correctly. In any case, many projects and standard libraries do use exceptions. It is therefore important to be familiar with them.

Source code for a version of the pay calculator program that includes the latest version of the `run_pay_calculator` function is available on the course web site under `PayCalculator2.5`.

### 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 two disadvantages of exceptions?

### Exercises

4.2.5. Revise the set methods of class `Time` so they throw exceptions of class `TimeError` in case their arguments are invalid. Define the class `TimeError` to be similar to the class `PayCalculatorError` we created in this section.

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 `PayCalculatorError` 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. But right now, the input operator of class `Time` doesn't generate an error as long as it reads an integer followed by any non-digit character and another integer. This means that the strings `8:5`, `37:50` and `3.14` would all be considered valid times.

In this section, we will fix some of that. We will have the `Time` input operator verify that the input consists of a valid numbers of hours, immediately followed by a colon and a valid number of minutes. This will mean that the strings `37:50` and `3.14` will now generate errors. (But `8:5` won't.)

In what follows, we will need to put a stream in the error state. (Recall the discussion of stream states in Section 3.3.) The state of a stream is determined by three flags that are stored in the stream. These flags are called the *eof bit*, the *fail bit* and the *bad bit*. The stream is in the eof state if the eof bit is set. The stream is in the error state if either the fail bit or the bad bit is set. The stream is in the good state if none of these bits are set.

Each of the state flags can set by using the method `setstate(stateflag)` 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 4.9 and 4.10 show a revised input operator for our class `Time`. Here's a decription of how it works (written in the first person).

We start by reading an integer using the standard input operator: `in >> h`. If that operation fails, the stream will be in the error state. In that case, there's nothing more we can do so we quit and return the stream as is. Otherwise, a valid integer was read so we check its value. If the value of `h` is invalid, we quit and return the stream after setting its fail bit.

We must then read a character, make sure it's a colon, check that there is no whitespace after the colon, read the number of minutes and check that this

```
istream & operator>>(istream & in, Time & t)
{
    // variables for reading the hours and minutes (so
    // we can leave t unchanged in case of an error)
    int h = 99;
    int m = 99;

    // read hours after skipping whitespace
    in >> h;
    if (!in) return in; // no int in stream
    if (h < 0 || h > 23) {
        in.setstate(in.failbit);
        return in;
    }

    // read next char and make sure it's a colon
    char next_char = in.get();
    if (!in) return in;
    if (next_char != ':') {
        in.putback(next_char);
        in.setstate(in.failbit);
        return in;
    }

    ...
}
```

Figure 4.9: Input operator that performs error checking (part 1 of 2)

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

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

    // read minutes
    in >> m;
    if (!in) return in;
    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(h, m);
    return in;
}
```

Figure 4.10: Input operator that performs error checking (part 2 of 2)

number is valid. And this really means that we must check that there is a character after the hours, make sure it's a colon, check that there is a character after the colon, make sure it's not whitespace, check that there is a number after the colon and make sure it's a valid number of minutes.

As mentioned earlier, the revised input operator of Figures 4.9 and 4.10 does not check that the hours and minutes consist of the correct number of digits. In addition, the current version of the input operator is *destructive*, in the sense that in case of an error, some data will be read from the stream and lost. Fixing both of these problems would require reading the characters one by one so they can be counted and restored in case of an error.

Note that we chose to have the input operator report errors through the 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. And the user may omit to test the stream for possible errors. Exceptions could be used to address both of these problems.

Source code for a class `Time` with the input operator presented in this section is available on the course web site under `PayCalculator2.5`.

## Study Questions

4.3.1. What method can be used to put a stream in the error state? What values can this method take as argument?

## 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 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 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 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. The operator should check that the hours and minutes have the specified number of digits and, in case of an error, no data should be lost.

# Chapter 5

# Vectors

In this chapter, we will learn how to use vectors to store large amounts of data.

## 5.1 A Simple File Viewer

We will illustrate the usefulness of vectors by designing and implementing a simple file viewer. This program will allow the user to view the contents of a text file.

Figure 5.1 shows what the interface of this program will look like. The program displays the name of the file that is currently being viewed, if any, followed by a certain number of lines from that file surrounded by a border. Below the text, a menu of commands is displayed followed by the prompt "`choice:`". The user types the first letter of a command, the command executes and everything is redisplayed.

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 to open.

We will design and implement the file viewer later in this chapter. The main

```
preface.txt
------------------------------------------------------------
  1  After a few computer science courses, students
  2  may start to get the feeling that programs can
  3  always be written to solve any computational
  4  problem.  Writing the program may be hard work.
  5  For example, it may involve learning a difficult
  6  technique.  And many hours of debugging.  But
  7  with enough time and effort, the program can be
  8  written.
  9
 10  So it may come as a surprise that this is not
 11  the case: there are computational problems for
 12  which no program exists.  And these are not
------------------------------------------------------------
  next  previous  open  quit
-------

command: o
file: introduction.txt
```

Figure 5.1: Sample interface of the file viewer

issue in the design of the program 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 idea would be as one very long string, with the different lines separated by new line characters. But then moving to the next or previous page would involve scanning the string character by character while counting new line characters. This is somewhat inefficient.

An alternative is to store each line on its own, as a string. We obviously can't use separate variables for each of the lines, since the number of lines is unknown and might be large. This is where vectors come in. In the next section, we describe what these are.

## 5.2 Vectors in the STL

In C++, a vector is a **container**, that is, a software component designed to hold a collection of elements. To be more precise, a **vector** holds a (finite) sequence of elements. These elements can be of any type but in an individual vector, all elements have to be of the same type. The type of element held by a particular vector is specified at the time that the vector is created.

C++ vectors are objects of class `vector`. This class is defined in library file `vector` and included in the `std` namespace.

Let's start with an example that illustrates some of the vector operations. Suppose that `v` is an empty vector of integers. This vector could have been created by using the default constructor as follows:

```
vector<int> v;
```

Note how the type of element stored in the vector is specified in its declaration.

Now suppose that we want to fill `v` with integers read from a file stream `in`. If the number of integers contained in the file is unknown, then this can be done as follows:

```
int x;
while (in >> x) v.push_back(x);
```

This loop reads every integer in the file and adds a copy of each one of them to the back of the vector. The size of the vector increases automatically.

Now suppose that the number of integers contained in the file is known to be `n`. In that case, it is more efficient to resize the vector once before the reading begins:

```
v.resize(n);
for (int & x : v) in >> x;
```

Note the difference: `v.push_back(x)` adds a new element to the vector while `in >> x` modifies an existing element of the vector.

In the above example, it is also possible to use the indexing operator:

```
v.resize(n);
for (int i = 0; i < n; ++i) in >> v[i];
```

But when a vector needs to be traversed in its entirety, as in this example, it is more convenient to use a range-for loop, as shown above.

Tables 5.1 and 5.2 show some of the most basic vector operations. The class supports several more, as described in a reference such as [CPP].[1]

The second constructor **value-initializes** the elements of the vector. A complete definition of value initialization involves concepts we have not covered but the following addresses the most common situations. If the vector elements are primitive values, such as **int**'s, **double**'s or **char**'s, then the elements are set to 0. (In the case of **char**, 0 is converted to the null character.) If the vector elements are objects, then they are initialized with their default constructor. If

[1] Some of these operations involve concepts that we 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({elements})`
`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` value-initialized elements of type `T`, or `n` copies of element `e`, or copies of the given `elements`, or copies of all the elements 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 value-initialized elements of type `T` or with copies of element `e`.

Table 5.1: Some basic vector operations (part 1 of 2)

`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.

`v1 = v2`
`v1 = {elements}`
: Makes vector `v1` contain copies of all the elements of vector `v2`, or copies of the given `elements`. Returns a reference to `v1`.

`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.clear()`
: Asks vector `v` to delete all its elements.

`v.assign(n, e)`
`v.assign({elements})`
: Asks vector `v` to change its contents to `n` copies of element `e`, or to copies of the given `elements`.

`v1.swap(v2)`
: Asks vector `v1` to swap contents with vector `v2` (without any elements being copied).

Table 5.2: Some basic vector operations (part 2 of 2)

the vector elements are arrays, then the elements of each of these arrays are value-initialized.[2]

The argument of the fourth constructor is an **initializer list**. When such a list is specified by using braces, the elements should be separated by commas as in `{1, 2, 3}`. In addition, the parentheses can be omitted:

```
vector<T> v{elements}
```

The following syntax can also be used:

```
vector<T> v = {elements}
```

The fifth constructor is known as a **copy constructor**. It can also be used with the equal sign syntax:

```
vector<T> v = v2
```

Note that 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.)

Vectors are said to be **dynamic** in the sense that they can grow and shrink as needed. Even then, vector do have a maximum size, which is related to the value of the largest integer that can be stored in an **`int`** variable. However, that maximum size, which can be retrieved by the method `max_size`, is typically

[2]One exception is when the vector elements are objects (or structures) that have no programmer-defined constructor. In that case, the objects are first **zero-initialized** and then initialized with the compiler-generated default constructor. When an object is zero-initialized, each of its data members is zero-initialized. If the data member is a primitive value, it is set to 0. If it's an array, all its elements are zero-initialized. The compiler-generated default constructor **default-initializes** each data member of the object. If the data member is a primitive value, nothing is done. If the data member is an object, it's initialized with its default constructor. If the data member is an array, all its elements are default-initialized.

much larger than what's needed in most applications. For example, for a vector of integers, a typical limit is approximately 1 billion elements.

Vectors are also said to be **generic** because they can hold elements of any type. In fact, vectors belong to a portion of the C++ standard library called the *Standard Template Library* (STL) and the word *template* in the name of this library refers to a C++ construct that allows the creation of generic software components. We will see later that the STL includes several other generic components, including other containers as well as functions that implement useful algorithms.

## Study Questions

5.2.1. What is a container?

5.2.2. In what way are vectors dynamic?

5.2.3. In what way are vectors generic?

5.2.4. What happens when a vector element is value-initialized?

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

## Exercises

5.2.6. 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.

5.2.7. Create a function `print(v, out)` that takes as arguments a vector of integers `v` and an output stream `out` and prints to `out` all the elements of `v`, one per line.

5.2.8. Create a function `concatenate(v1, v2)` that takes as arguments two vectors of integers `v1` and `v2` and returns another vector that contains a copy of all the elements of `v1` followed by a copy of all the elements of `v2`.

5.2.9. Create a function `replace(v, x, y)` that takes as arguments a vector of integers `v` and two other integers `x` and `y`, and replaces every occurrence of `x` in `v` by a copy of `y`.

5.2.10. Create a function `count(v, x)` that takes as arguments a vector of integers `v` and an integer `x` and returns the number of occurrences of `x` in `v`.

## 5.3 Design and Implementation of the File Viewer

The general behavior of the file viewer was described in the previous section but several details still need to be specified. For example, how does the program know how many lines to display per "page"? What should the `next` command do when the last page is already displayed? How should the last page be displayed if it is shorter than the others? What should happen if a file doesn't open? A full specification of the file viewer should answer these questions as well as every other possible question about the behavior of the program. One possible specification is shown in Figures 5.2 to 5.4.

We now turn to the design of the program. At any moment in time, the file viewer holds a copy of the contents of a file. We will call this copy a *buffer*. There are three main tasks that the file viewer needs to handle:

1. The user interaction.

2. The storage of the buffer.

3. The execution of the commands.

```
OVERVIEW

A simple file viewer that allows the user view the
contents of a text file.


DETAILS

The program interacts with the user as shown in the
following example:

preface.txt
──────────────────────────────────────────────────────
  1  After a few computer science courses, students
  2  may start to get the feeling that programs can
  3  always be written to solve any computational
  4  problem.  Writing the program may be hard work.
  5  For example, it may involve learning a difficult
  6  technique.  And many hours of debugging.  But
  7  with enough time and effort, the program can be
  8  written.
  9
 10  So it may come as a surprise that this is not
 11  the case: there are computational problems for
 12  which no program exists.  And these are not
──────────────────────────────────────────────────────
  next  previous  open  quit
──────
command: o
file: introduction.txt
```

Figure 5.2: Specification of the file viewer (part 1 of 3)

```
The program begins by asking the user for a window
height.  This is the number of lines that will be
displayed as each "page".  The displayed lines are
numbered starting at 1 for the first line of the file.
 If the number of lines on the last page is smaller
than the window height, the rest of the window is
filled with unnumbered empty lines.

Each page is displayed between two lines of 50 dashes.
 The name of the file is printed above the first line
of dashes.  Below the second line of dashes, a menu of
commands is displayed.  Below that menu, the prompt
"choice:" is displayed.  The user types the first
letter of a command, the command executes and
everything is redisplayed.  Some commands prompt the
user for more information.

Here is a description of the various commands:

next: The next page is displayed.  Does nothing if the
last line of the file is already displayed.

previous: The previous page is displayed.  Does
nothing if the first line of the file is already
displayed.

open: Asks for a file name (with prompt "file:") and
displays that file.  If a file named X does not open,
the message "ERROR: Could not open X" is displayed
just before the file name is redisplayed.
```

Figure 5.3: Specification of the file viewer (part 2 of 3)

```
quit: Stops the program.


NOTES FOR LATER VERSIONS

Add more error-checking.  (Check that commands are
entered properly and that the window height is a
positive integer.)
```

Figure 5.4: Specification of the file viewer (part 3 of 3)

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 program, including most of the user interaction, is assigned to a `FileViewer` component.

Here are some more details on the program's design. The `FileViewer` has a single public method, as shown in Figure 5.5. The `FileViewer` stores the lines of text in a `Buffer` object. It also delegates the displaying of the text and the execution of the commands to that `Buffer` object. Accordingly, the `Buffer` class has several public methods, including one for displaying the buffer and one for each of the file viewer commands, as shown in Figure 5.6.

We now examine the implementation of the program. The private `FileViewer` method `display` is a helper method to which `run` delegates the displaying of the buffer and of the menu of commands. Its implementation is shown in Figure 5.7. The data member `error_message` is set when an error occurs during the execution of a command. In this version of the file viewer, the only error that can occur is that a file doesn't open.

Figure 5.8 shows most of the implementation of the `FileViewer` method `run`. Figure 5.9 shows the body of the **`switch`** statement. Figures 5.10 and 5.11 show the implementation of the `Buffer` methods that weren't already imple-

```
class FileViewer
{
public:
    void run();

private:
    void display();

    Buffer buffer;
    int window_height;
    std::string error_message;
};
```

Figure 5.5: Declaration of `FileViewer`

mented in the class declaration. All of this code is fairly straightforward.

The complete source code and documentation of the file viewer is available on the course web site under `FileViewer`.

## Exercises

5.3.1. Modify the file viewer as described below. Change the original specification, design and implementation as little as possible. Error messages should be displayed at the top of the window, just like when a file does not open.

a) Add a *jump* command that asks the user for a line number and redisplays the file with the requested line at the top. In case the user enters an invalid line number *N*, the program should print the error message *ERROR: N is not a valid line number*. (You'll need to revise

```
class Buffer
{
public:
    void display() const;
    const std::string & get_file_name() const
    {
        return file_name;
    }
    void move_to_next_page();
    void move_to_previous_page();
    bool open(const std::string & file_name);
    void set_window_height(int h)
    {
        window_height = h;
    }

private:
    std::vector<std::string> v_lines;
    int ix_top_line = 0;
    std::string file_name;
    int window_height;
};
```

Figure 5.6: Declaration of `Buffer`

```
void FileViewer::display()
{
    const string long_separator(50, '-');
    const string short_separator(8, '-');

    if (!error_message.empty()) {
        cout << "ERROR: " + error_message << endl;
        error_message.clear();
    }

    string file_name = buffer.get_file_name();
    if (file_name.empty())
        cout << "<no file opened>\n";
    else
        cout << file_name << endl;

    cout << long_separator << endl;
    buffer.display();
    cout << long_separator << endl;
    cout << "  next  previous  open  quit\n";
    cout << short_separator << endl;
}
```

Figure 5.7: The `FileViewer` method `display`

```
void FileViewer::run()
{
    cout << "Window height? ";
    cin >> window_height;
    cin.get();  // '\n'
    cout << '\n';
    buffer.set_window_height(window_height);

    bool done = false;
    while (!done) {
        display();

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

        switch (command) {
            ...
        };
        cout << endl;
    } // while

    return;
}
```

Figure 5.8: The `FileViewer` method `run`

```
switch (command) {
    case 'n': {
        buffer.move_to_next_page();
        break;
     }

    case 'o': {
        cout << "file name: ";
        string file_name;
        getline(cin, file_name);
        if (!buffer.open(file_name))
            error_message =
                "Could not open " + file_name;
        break;
    }

    case 'p': {
        buffer.move_to_previous_page();
        break;
    }

    case 'q': {
        done = true;
        break;
    }
};
```

Figure 5.9: The **switch** statement of `run`

```
inline void Buffer::move_to_next_page()
{
    if (ix_top_line + window_height < v_lines.size())
        ix_top_line += window_height;
}

inline void Buffer::move_to_previous_page()
{
    if (ix_top_line > 0) ix_top_line -= window_height;
}

void Buffer::display() const
{
    int ix_stop_line = ix_top_line + window_height;
    for (int i = ix_top_line; i < ix_stop_line; ++i) {
        if (i < v_lines.size())
            cout << std::setw(6) << i+1 << "  "
                 << v_lines[i];
        cout << '\n';
    }
}
```

Figure 5.10: Implementation of some of the `Buffer` methods

```
bool Buffer::open(const string & new_file_name)
{
    std::ifstream file(new_file_name);
    if (!file) return false;

    v_lines.clear();
    // Note: the vector is cleared only after we know
    // the file opened successfully.

    string line;
    while (getline(file, line))
        v_lines.push_back(line);

    file_name = new_file_name;
    ix_top_line = 0;
    return true;
}
```

Figure 5.11: Implementation of the `Buffer` method `open`

the *previous* command to ensure that it doesn't move past the beginning of the file after a jump.)

b) Add a *search* command that asks the user for a string and finds the first line that contains that string. The search starts at the first line currently displayed and stops at the end of the file. If the string is found, the line that contains it is displayed at the top. In case the user enters a string *X* that does not occur in the file, the program should print the error message *ERROR: string "X" was not found*.

c) Modify the *search* command so that the search wraps around to the beginning of the file when the end of the file is reached.

d) Add an *again* command that repeats the last search. If that last search was the previous command, then the new search starts at the second line that is currently displayed. If no search has been previously performed, then the *again* command asks the user for a string.

5.3.2. Add more error checking to the file viewer. Change the program as little as possible.

a) Make the program check that the user enters a positive integer as the window height. If not, the program should print an error message and ask again.

b) Make the program check that the user enters either the whole name of a command or the first letter of the command. If the user enters an invalid command *X*, the program should print the error message *ERROR: X is not a valid command*. The error message should be displayed at the top of the window, just like when a file does not open.

## 5.4 Vectors and Exceptions

As mentioned earlier in these notes, some components of the C++ standard library use exceptions. Vectors are one example.

There are two types of exceptions that vectors may throw. One concerns the allocation of memory. In principle, any operation that causes a vector to increase the amount of memory it uses will throw a `bad_alloc` exception if there isn't enough memory available to the program. This includes the `resize`, `push_back` and `assign` methods, as well as the assignment operator and the various constructors.

The second type of exception a vector may throw concerns access to individual elements. The context here is that the vector indexing operator is not safe: it does not check that the given index is valid, which may cause the operator to access memory that does not belong to the vector. This can cause the program to crash. Perhaps even worse, it can cause an incorrect value to be retrieved or the value of some other variable to change. These errors can introduce into a program bugs that are very hard to find.

This lack of safety means that it's up the user of the indexing operator to verify that indices are valid. In most circumstances, this is desirable because it allows the indexing operator to execute more quickly.

On the other hand, if in some context a safe indexing operator is preferable, vectors provide a method `at` that takes an index as argument and returns a reference to the corresponding element, just like the indexing operator. But, unlike the indexing operator, the `at` method will throw an `out_of_range` exception if the index is invalid.

The `bad_alloc` exception class is defined in the `new` library. The `out_of_range` exception class is defined in the `stdexcept` library. Both classes are part of the `std` namespace.

`T a[N]`
`T a[N] = {elements}`
`T a[] = {elements}`
: Creates an array `a` that can hold elements of type `T`. The array is initialized to contain `N` default-initialized elements of type `T`, or copies of the given `elements`. In case the number of given `elements` is less than `N`, the remaining elements of `a` are value-initialized. `N` must be a compile-time constant.

`a[i]`
: Returns a reference to the element at index `i` in array `a`.

Table 5.3: Some basic array operations

## 5.5 Arrays

In this chapter, we learned that C++ vectors can be used to store sequences of elements. But there is a simpler alternative: ordinary arrays. Table 5.3 shows how arrays can be created and how their elements can be accessed.

In the first form of array declaration, the elements of the array are *default-initialized*. Recall that this means that if the array elements are primitive values, then the elements are not initialized. If the array elements are objects, then they are initialized with their default constructor. If the array elements are themselves arrays, then the elements of each of these arrays are default-initialized.

In the second form of array declaration, elements that are not given explicit initial values are *value-initialized*. Recall that this means that if these elements are primitive values, then the elements are set to 0. If these elements are objects, then they are initialized with their default constructor. If these elements are themselves arrays, then the elements of each of these arrays are value-initialized. (See the footnote in Section 5.2 for an exception.)

Ordinary C++ arrays have one advantage over vectors: when used in some contexts, the array indexing operator runs slightly more quickly than the vector indexing operator.

However, ordinary arrays have several major weaknesses compared to vectors. In fact, vectors were developed precisely to address these weaknesses.

The first weakness, and perhaps the main one, is that ordinary C++ arrays have a predetermined size. Predetermined here 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.

In contrast, as we already know, vectors can be declared to be of any size and that size can be changed as needed during the execution of the program, by using methods such as `resize`, `push_back` and `assign`.

Later in these notes, when we learn the techniques that are used to implement containers such as vectors, we will learn that arrays can be *dynamically allocated*. However, this involves low-level techniques that are somewhat inconvenient and prone to errors. In any case, all C++ arrays, even those that are dynamically allocated have several other weaknesses.

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 5.12 shows a function that prints the contents of an array of integers. One danger is that nothing guarantees that the value of the size argument is correct. In contrast, vectors are aware of their size and whenever we need to know that size, we just have to ask by using the method `size()`.

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

```
void print(const int a[], int n)
{
    for (int i = 0; i < n; ++i)
        cout << a[i] << '\n';
}
```

Figure 5.12: A function that prints an array

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. In contrast, vectors support all the usual operators, including =, == and <.

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. It is possible to get around this problem by using dynamically allocated arrays. But, once again, these techniques are inconvenient and prone to errors. In contrast, functions can return copies of vectors just as if they were a value of any of the primitive data types.[3]

To summarize, ordinary C++ arrays have the following four weaknesses: (1) they have a fixed, predetermined size, (2) they don't know their size, (3) they don't support the usual operators and (4) functions cannot easily return copies of arrays.

## Study Questions

5.5.1. What are two advantages of arrays over vectors?

5.5.2. What are four advantages of vectors over ordinary arrays?

[3]Note, however, that this is something that is usually done only with small vectors. It is more efficient to avoid the copying of the vector by passing it to the function as a reference argument.

## Exercises

5.5.3. Modify the file viewer by using an array to implement the `Buffer`. Use an array of size 100,000. In case a file called *X* is too large, the program should print the error message *ERROR: file X is too large* and redisplay the previous file.

5.5.4. Repeat the previous exercise but this time, in case the file is too large, have the program use the array to store part of the file. When another part is needed, the program reopens the file to read that part and store it in the array.

# Chapter 6

# Generic Algorithms

The C++ Standard Template Library includes algorithms that solve a wide variety of common problems. In this chapter, we will learn how to use and implement some of these algorithms. In the process, we will become familiar with the important concepts of iterators, function templates, function objects and generic programming.

## 6.1 Introduction

Some problems come up very frequently in a wide variety of software projects. To illustrate, here are just three examples:

1. Computing the maximum of two values.

2. Counting the number of occurrences of an element in some container (a vector, for example).

3. Sorting the elements of a container in some order.

Standard algorithms have been developed for all of these common problems. Some are trivial: to compute the maximum of two values, simply compare them by using the less-than operator (<). Others are nontrivial but simple: to count the number of occurrences of an element in a sequence container such as a vector, initialize a counter to 0 then scan the sequence from beginning to end while adding one to the counter every time an occurrence of the element is seen. Some of these algorithms, on the other hand, can be fairly complex. This includes efficient algorithms for sorting vectors.[1]

When needed, we can always implement these standard algorithms ourselves. But the STL includes functions that implement many of these algorithms. For example,

```
max(a, b)
```

returns the maximum of `a` and `b`. The number of occurrences of element `e` in vector `v` can be computed as follows:

```
count(v.begin(), v.end(), e)
```

And

```
sort(v.begin(), v.end())
```

rearranges the elements of vector `v` in increasing order. These functions are called *STL algorithms*.

Using STL algorithms has a number of advantages compared to implementing algorithms ourselves. First, obviously, it saves time. Second, it leads to more reliable software because the STL algorithms have been more extensively tested than any of our own implementations ever could. Third, it makes our programs easier to understand because other programmers are already familiar with the STL algorithms.

[1] Examples are mergesort, heapsort and quicksort. We will study mergesort and quicksort later in these notes. At Clarkson, heapsort is normally covered in the course *CS344 Algorithms and Data Structures*.

The use of STL algorithms, as well as the use of any software component from a standard library, is an example of *software reuse*. Reusing any software component, but especially standard components, generally gives those three benefits: faster development, increased reliability, and software that's easier to understand.

Note that the STL algorithms are **generic**, in the sense that they can be used on arguments of more than one type. For example, `max` can be used on any type of value that supports the less-than operator (<). The algorithm `count` can be used on any type of vector, as long as the element type supports the equality testing operator (==). And `sort` can be used on any vector whose element type supports the less-than operator (<).

Now, in the above examples, exactly what type of value are the arguments `v.begin()` and `v.end()`? This is what we will learn in the next section.

## Study Questions

6.1.1. What are three benefits of using standard software components such as the STL algorithms?

6.1.2. What is a generic algorithm?

## Exercises

6.1.3. Experiment with the three STL algorithms mentioned in this section by writing a test driver that uses each one of them on more than one type of argument. (These algorithms are defined in the library file `algorithm` and included in the `std` namespace.)

## 6.2 Iterators

In the previous section, we learned that the number of occurrences of element `e` in vector `v` can be computed as follows by using the STL algorithm `count`:

```
count(v.begin(), v.end(), e)
```

The first two arguments specify the portion of the vector over which the counting occurs. But exactly what type of value are those arguments?

These arguments are **iterators**. In fact, the STL algorithm `count` has the general form

```
count(start, stop, e)
```

where the arguments `start` and `stop` are iterators that specify the positions in a container where the counting will start and where it will stop.

All STL algorithms use iterators to specify positions. Another example is the `sort` algorithm we mentioned in the previous section:

```
sort(start, stop)
```

sorts the elements that occur in the portion of a container that begins at `start` and ends at `stop`.

We will soon learn exactly what iterators are and what we can do with them. But, first, why do STL algorithms use iterators and not indices to specify positions within a container? The answer is simple: indices are not an efficient way of accessing elements in most types of containers, including the `list` and `map` containers that we will study later in these notes. For this reason, most types of containers don't support indices (that is, they don't have an operation like an indexing operator that allows an element to be accessed given its index). Iterators, on the other hand, can be used to efficiently access elements in most types of containers, including vectors, lists and maps.

But then, why don't STL algorithms come in two versions, one for iterators and one for indices? For example, the index version of `count` could have the general form

```
count(container, i, j, e)
```

where the arguments `i` and `j` are indices that specify where the counting starts and stops. One possible answer is that since iterators can be used to efficiently access elements in a vector, the index version of `count` is not needed. By not including it, the STL is kept simpler.

Perhaps a better answer relates to a design issue. When writing code that involves vectors, it is sometimes necessary to use indices. But sometimes iterators would also do the job perfectly well. And in those situations, it is better to use iterators because the same code can later be reused with other containers, such as lists, that don't support indices. The fact that the STL only includes iterator versions of its algorithms encourages us to do precisely that: to use iterators, whenever possible, because this leads to code that's more general and has a greater chance of being reused.

So what exactly is an iterator? An iterator is a value, usually an object, whose most basic purpose is simply to mark a position within a container. We say that the iterator *points* to the element that occupies that position.

For example, consider

```
count(v.begin(), v.end(), e)
```

As we said earlier, this returns the number of occurrences of `e` in `v`. The `begin` method returns a *begin iterator*, one that points to the first element of the vector. The `end` method returns a special iterator called the *end iterator*. This iterator does not point to any elements. Instead, it should be thought of as pointing to a position that's just past the last element of the container. This is consistent with the fact that the second argument of `count` should be the position where the counting stops.

In general,

```
count(start, stop, e)
```

counts from `start` to `stop`. In this context, the pair of iterators `start` and `stop` specifies a *range of positions*. In the STL, ranges of positions are always

specified by two iterators that act as the endpoints of an interval. The first iterator is included in the range but the second one is not. Such a range can be represented using the usual mathematical notation as [`start`, `stop`).

Now, let's say that we want to count the number of occurrences of `e` among the first 10 elements of `v`. To specify that range of positions, we need an iterator that points to `v[0]` and one that points to `v[10]`. The begin iterator points to `v[0]`. An iterator that points to `v[10]` can be obtained by adding 10 to `v.begin()`:

```
count(v.begin(), v.begin() + 10, e)
```

This works because, in general, `itr + i` produces an iterator that points `i` positions to the right of `itr`. Note that this gives us a way to convert indices into iterators.

It's also possible to do the reverse conversion, from a vector iterator to an index. This can be used, for example, with STL algorithms that return iterators. One of those is the `find` algorithm. For example,

```
find(v.begin(), v.end(), e)
```

returns an iterator to the first occurrence of `e` in vector `v`. That iterator can then be converted to an index by computing the difference between the iterator and the begin iterator:

```
find(v.begin(), v.end(), e) - v.begin()
```

This works because the subtraction operator returns the distance, in number of elements, between the two iterators. Since `v.begin()` points to the element with index 0, the distance `itr - v.begin()` is the index of the element that `itr` points to.

In general,

```
find(start, stop, e)
```

returns an iterator to the first occurrence of `e` in the range [`start`, `stop`). In case `e` is not found in that range, `find` returns `stop`. In the previous example, `find` would return `v.end()`. To detect that possibility, we could convert the returned iterator to an index and test if it equals `v.size()`:

```
if (find(v.begin(), v.end(), e) - v.begin() ==
        v.size()) ...
```

This works because if the end iterator pointed to an element, that element would have index `v.size()`. But it's simpler to directly compare the iterator returned by `find` to `v.end()`:

```
if (find(v.begin(), v.end(), e) == v.end()) ...
```

And this method has the advantage of working on all types of containers, not just containers that have indices.

As we said earlier, the most basic purpose of an iterator is to mark a position in a container. But an iterator can also be used to access the element at that position. For example, suppose that we need to change to 12 the first occurrence of 5 in vector `v`. This could be done by using `find` to locate the first 5 and then converting the iterator into an index:

```
int i = find(v.begin(), v.end(), 5) - v.begin();
v[i] = 12;
```

But it is simpler to use the iterator directly:

```
*find(v.begin(), v.end(), 5) = 12;
```

In this code, the *dereferencing operator* (`*`) is applied to the iterator returned by `find`. In general, if `itr` is an iterator, then `*itr` refers to the element that `itr` points to.

In the previous example, we might prefer to first store the iterator returned by `find` and then dereference it:

```
auto itr = find(v.begin(), v.end(), 5);
*itr = 12;
```

The keyword **auto** asks the compiler to deduce the appropriate type for the variable. In this case, `itr` will be given the type of the iterator returned by `find`.

The use of **auto** is a good idea here because iterator types are a bit messy. For example, if `v` is a (non-constant) vector of integers, then `find` in the above example would return an iterator of type

```
vector<int>::iterator
```

This is because each type of STL container provides its own type of iterator and that type is defined within the class itself. The double-colon (`::`) is used to access that type. In general, if `Container` is a container type, then

```
Container::iterator
```

is the type of iterator that works with `Container`.

Note that **auto** should only be used where it really improves readability. That's the case with complex types such as `vector<int>::iterator` but not with simple types such as **int** and `string`. In those cases, it is better to specify the type explicitly.

Iterators can also be used to step through, or iterate over, the elements of a container. (That's why they're called iterators. In other languages, sometimes they're called enumerators.) For example, suppose that we need to print all the elements of vector `v` starting at the first occurrence of 5. This could be done with indices but, once again, it is simpler to use iterators directly, as shown in Figure 6.1. The iterator `itr` is initialized to the position of the first 5. At each iteration of the loop, the *increment operator* (++) is used to make the iterator point to the next element. Eventually, the iterator will move past the last element, to the position of the end iterator. This will cause the loop to terminate.

```
for (auto itr = find(v.begin(), v.end(), 5);
        itr != v.end();
        ++itr)
    cout << *itr << ' ';
```

Figure 6.1: Using iterators to traverse a container

To summarize, in C++ and many other programming languages, iterators are a uniform and efficient way of specifying positions and accessing elements in many types of containers. An iterator is a value (usually an object) that marks a position in a container, provides access to the element at that position, and allows us to step through all the elements of a container. In the STL, positions are specified by using iterators. This allows STL algorithms to be highly generic in that they can be used on different types of containers.

All standard types of iterators support the basic operations $*$, ++, == and !=. Vector iterators support additional operations, as described in this section. In the next section, we will describe in more detail the operations supported by various types of iterators.

## Study Questions

6.2.1. What is an iterator?

6.2.2. In what way do iterators allow algorithms to be more generic?

6.2.3. Does it make sense to dereference the end iterator of a container?

6.2.4. What is the C++ keyword **`auto`** used for?

6.2.5. What are some of the operations supported by vector iterators?

6.2.6. How is a range of positions specified in the STL?

## Exercises

6.2.7. Assuming that `v` is a vector of integers, write code fragments that perform the following tasks. Use the STL algorithms `count` and `find`.

a) Set integer variable `count` to the number of 0's in the first half of `v`. (In case `v` is of odd length, consider that the middle element is in the second half of `v`.)

b) Change to 1 the first 0 that occurs in `v`. Do nothing if the vector does not contain any 0's.

c) Add 1 to every element that precedes the first 0 in `v`. If `v` contains no 0's, add 1 to every element in `v`.

6.2.8. Create the following functions, which are intended to be used on vectors of integers. In each case, the arguments `start` and `stop` are iterators of type `vector<int>::iterator`. The arguments `e`, `x` and `y` are integers.

a) `count(start, stop, e)` returns the number of occurrences of element `e` in the range [`start`, `stop`).

b) `fill(start, stop, e)` sets to `e` every element in the range [`start`, `stop`).

c) `find(start, stop, e)` returns an iterator to the first occurrence of element `e` in the range [`start`, `stop`). In case `e` is not found, `stop` is returned.

d) `replace(start, stop, x, y)` replaces by `y` every occurrence of element `x` in the range [`start`, `stop`).

(Note that these functions are special cases of the STL algorithms that have the same name. These special cases work only on vectors of integers but later in this chapter, we will learn to create functions that can be used on multiple types of containers, just like the STL algorithms.)

## 6.3 Iterator Types and Categories

In this section, we will learn that STL containers each provide several types of iterators. In addition, each iterator type belongs to a *category* that determines what operations the iterators of that type support.

Consider the code

```
find(v.begin(), v.end(), e)
```

If `v` is a non-constant vector, then `v.begin()` and `v.end()` return iterators of type `iterator`, which causes `find` to return an iterator of the same type. As mentioned in the previous section, this type of iterator can be used to modify the element that it points to.

If, on the other hand, `v` is a constant vector, then `v.begin()` and `v.end()` return iterators of type `const_iterator`, which once again causes `find` to return an iterator of the same type. These are called **constant iterators** because they cannot be used to modify the element they point to. This makes sense because it prevents the elements of a constant vector from being modified.

Now, suppose that `v` is a non-constant vector but that for some reason we do not wish to modify it. Then, in the above code, we could instead use the methods `cbegin` and `cend`:

```
find(v.cbegin(), v.cend(), e)
```

These methods always return constant iterators, even if the vector is non-constant. This would cause `find` to return a constant iterator, thereby protecting the vector elements from modification.

In addition to plain iterators and constant iterators, STL containers provide **reverse iterators** and **constant reverse iterators**. These iterators are of type `reverse_iterator` and `const_reverse_iterator`, respectively.

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` that returns a reverse iterator that points to the *last* element of the

container and a method `rend` that returns a reverse iterator that points to a position *just before* the first element of the container. In addition, the ++ operator works in reverse direction with reverse iterators. The methods `rbegin` and `rend` also have versions `crbegin` and `crend` that always return constant reverse iterators.

For example, the following code prints the contents of vector `v` in reverse order:

```
for (auto itr = v.crbegin();
        itr != v.crend();
        ++itr)
    cout << *itr;
```

And

```
find(v.rbegin(), v.rend(), e)
```

searches `v` in reverse order, therefore returning an iterator to the last occurrence of `e`.

So iterators can be of four different types: plain, constant, reverse, constant reverse. But there are also several iterator **categories**. The two most important ones are *bidirectional iterators* and *random-access iterators*.

The difference between these iterator categories is in the set of operations that they support. **Bidirectional iterators** support the basic operations `*`, `->`, `++`, `--`, `==` and `!=` described in Table 6.1.

The operator `->` is called the **dereference-and-select operator** or, more simply, the *arrow operator*. It is a convenient combination of the dereferencing operator (`*`) followed immediately by the selection operator (`.`). For example, if `itr` points to a `Time`, then the message `hours` can be sent to `*itr` as either `(*itr).hours()` or `itr->hours()`. The arrow version is easier to both write and read.

**Random-access iterators** support all the operations supported by bidirectional iterators plus the following ones: `+i`, `-i`, `+=i`, `-=i`, `-`, `<` and `[i]`. These

| |
|---|
| `*itr`<br>Returns a reference to the element that `itr` points to. |
| `itr->f(args)`<br>`itr->var`<br>Accesses 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. |
| `++itr`<br>`--itr`<br>Makes `itr` point to the next or previous element. |
| `itr1 == itr2`<br>`itr1 != itr2`<br>Returns **`true`** if `itr1` and `itr2` point or don't point to the same element. |

Table 6.1: Operations supported by bidirectional iterators

`itr + i`
`itr - i`
: Returns an iterator that points `i` positions to the right or left of `itr`.

`itr += i`
`itr -= i`
: Moves the iterator `i` positions to the right or left.

`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 6.2: Additional operations supported by random-access iterators

operations are described in Table 6.2. Note that `itr[0]` is equivalent to `*itr` and that, in general, `itr[i]` is equivalent to `*(itr + i)`.

Vector iterators are random-access iterators. C++ strings also provide random-access iterators. Lists and maps, two containers we will study later in these notes, only provide bidirectional iterators.

## Study Questions

6.3.1. What is the difference between a plain iterator and a constant iterator?

6.3.2. What method always returns a constant begin iterator?

6.3.3. What position does a reverse begin iterator point to? What about a reverse end iterator?

6.3.4. In what direction does the —— operator move a reverse iterator?

6.3.5. What operations are supported by bidirectional iterators?

6.3.6. What operations are supported by random-access iterators?

### Exercises

6.3.7. Experiment with the operations of random-access iterators by writing a test driver that uses all the operations shown in Table 6.2.

## 6.4 Vectors and Iterators

We now know that vectors provide random-access iterators. We also know that the methods `begin`, `end` and their variations can be used to obtain vector iterators. But there are several other vector operations that take iterators as arguments. Table 6.3 shows some of them.

Note that several vector operations may invalidate existing vector iterators. In general, whenever the size of a vector increases, all iterators may become invalid. When elements are removed from a vector, then the following iterators are invalidated: all iterators that point to the removed elements or to positions to the right of those elements, including the end iterator.

### Study Questions

6.4.1. What operations may invalidate vector iterators?

`vector<T> v(start, stop)`
: Creates a vector `v` that can hold elements of type `T`. The vector is initialized with copies of all the elements in the range [`start`, `stop`).

`v.assign(start, stop)`
: Asks vector `v` to change its contents to a copy of all the elements in the range [`start`, `stop`).

`v.insert(itr, e)`
: Asks vector `v` 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.

`v.insert(itr, {elements})`
`v.insert(itr, start, stop)`
: Asks vector `v` to insert, at the position indicated by the iterator `itr`, copies of the given `elements`, or copies of all the elements in the range [`start`, `stop`). An iterator that points to the first new element is returned. If the range is empty, `itr` is returned.

`v.erase(itr)`
: Asks vector `v` to delete the element that `itr` points to. An iterator that points to the next element is returned.

`v.erase(start, stop)`
: Asks vector `v` to delete all the elements in the range [`start`, `stop`). An iterator that points to the next element is returned.

Table 6.3: Some vector operations that involve iterators

### Exercises

6.4.2. Experiment with the vector operations that use iterators by writing a test driver that uses all the operations shown in Table 6.3.

## 6.5 Algorithms in the STL

This section presents several of the generic algorithms available in the STL. Many more are described in a reference such as [CPP].

Tables 6.4 and 6.5 list some generic algorithms that are designed to be used on sequences such as vectors. All of these algorithms work by traversing the sequence from one end to the other. Accordingly, their running time is proportional to the number of elements in the range. More formally, we say the running time of these algorithms is $\Theta(n)$ where $n$ is the number of elements in the range.

We will define the notation $\Theta(n)$ precisely later in these notes. For now, we will only consider its intuitive meaning: a running time is $\Theta(n)$ when it is essentially proportional to $n$. Such a running time is called *linear*.

The `find` generic algorithm performs what is called a sequential search of the range. In case the elements in the range are sorted, the *binary search* algorithm runs much faster, as long as elements can be accessed quickly.

The STL includes several versions of the binary search algorithm. These are described in Table 6.6. When given random-access iterators, these algorithms run in time $\Theta(\log n)$. (That's the log in base 2 of $n$.) This is called logarithmic time. Since $\log 10^6$ is about 20, in principle, we can expect a binary search of a vector of size one million to run about 50,000 times faster than a sequential search. That would be the difference between one millisecond and almost one minute.

Table 6.6 also includes a description of the generic algorithm `sort` we mentioned earlier in this chapter. This algorithm requires random-access iterators and runs in time $\Theta(n \log n)$. In comparison, the simplest sorting algorithms all run in time $\Theta(n^2)$. Once again, in principle, we can expect the generic algorithm

`count(start, stop, e)`
: Returns the number of occurrences of element `e` in the range `[start,stop)`. Uses == on elements.

`find(start, stop, e)`
: 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.

`max_element(start, stop)`
`min_element(start, stop)`
: Returns an iterator that points to the maximum or minimum element in the range `[start,stop)`. Returns `stop` if the range is empty. Uses < on elements.

`copy(start1, stop1, start2)`
: Copies the elements in the range `[start1,stop1)` to a range of positions that begins at `start2`. If *n* is the number of elements that are copied, returns an iterator that points *n* positions to the right of `start2`. Copies forward, from `start1` to `stop1`. Typically not useful when `start2` falls within the range `[start1,stop1)`.

`copy_backward(start1, stop1, stop2)`
: Copies the elements in the range `[start1,stop1)` to a range of positions immediately to the left of `stop2`. If *n* is the number of elements that are copied, returns an iterator that points *n* positions to the left of `stop2`. Copies backward, from `stop1` to `start1`. Normally used instead of `copy` when `start2` falls within `[start1,stop1)`.

Table 6.4: Some generic sequence algorithms (part 1 of 2)

`fill(start, stop, e)`
: Sets all the elements in the range `[start,stop)` to be copies of element `e`.

`replace(start, stop, x, y)`
: Replaces all occurrences of element `x` by a copy of element `y` in the range `[start,stop)`. Uses == on elements.

`reverse(start, stop)`
: Reverses the order of the elements in the range `[start,stop)`.

Table 6.5: Some generic sequence algorithms (part 2 of 2)

`sort` to run about 50,000 faster than these simple sorting algorithms on a vector of size one million. That would be the difference between one second and about 14 hours. (We will study searching and sorting algorithms in more detail later in these notes.)

Table 6.7 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 ($\Theta(1)$). When used with bidirectional iterators, the algorithms run in linear time.

The generic algorithm `swap` is defined in the library file `utility`. The generic algorithms `advance` and `distance` are defined in the library file `iterator`. The other STL generic algorithms are all defined in the library file `algorithm`. All of them are part of the `std` namespace.

`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. All versions run in time $\Theta(\log n)$ if the arguments are random-access iterators. If the arguments are only bidirectional iterators, the running time is $\Theta(n)$.

`sort(start, stop)`
: Sorts the elements in the range `[start,stop)` using the < operator. Requires random-access iterators. Runs in time $\Theta(n \log n)$.

Table 6.6: Some generic algorithms for searching and sorting

`swap(x, y)`
: Swaps the values of `x` and `y`.

`max(x, y)`
`min(x, y)`
`max({elements})`
`min({elements})`
: Returns the maximum or minimum of the given elements. Uses $<$ to compare elements.

`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 and therefore runs in time $\Theta(n)$.

`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`, which gives a $\Theta(n)$ running time. Assumes that `itr2` is reachable from `itr1`.

Table 6.7: Some other generic algorithms

### Study Questions

6.5.1. What is the intuitive meaning of $\Theta(n)$?

6.5.2. How does the running time of a binary search compare to that of a sequential search?

6.5.3. What is the running time of the generic algorithm `sort`?

### Exercises

6.5.4. Explain why the generic algorithm `copy` is typically not useful when `start2` falls within the range `[start1,stop1)`?

6.5.5. Experiment with the STL searching and sorting algorithms by writing a test driver that uses all the algorithms shown in Table 6.6. To see the difference between `lower_bound` and `upper_bound`, make sure you search for an element that occurs multiple times in the sequence.

## 6.6 Implementing Generic Algorithms

In this section, we will learn how to implement generic algorithms. Recall that this means algorithms that can be used on arguments of more than one type.

Consider the `max` generic algorithm. Figure 6.2 shows an implementation of an integer version of this algorithm. If ever we needed a version for another type of argument, all we would have to do is replace the three occurrences of **`int`** by the other type. For example, Figure 6.3 shows a `string` version.

To simplify this process, and to reduce the risk of errors, we could explicitly identify the types that must be changed to produce a new version of the algorithm, as shown in Figure 6.4. The result is a **template** in which the generic name `T` is used for the type of value being compared. To make things clear, the template definition begins with a declaration that identifies `T` as a **template**

```
inline const int & max(const int & x, const int & y)
{
    return (x < y ? y : x);
}
```

Figure 6.2: An integer version of the `max` algorithm

```
inline const string & max(const string & x,
                          const string & y)
{
    return (x < y ? y : x);
}
```

Figure 6.3: A `string` version of the `max` algorithm

**parameter**. Then, whenever a version of `max` is needed for a particular type of value, all we have to do is copy the template and replace every occurrence of `T` by the desired type.

What we just described is a fairly mechanical process that's a good candidate for automation. And, in fact, C++ compilers can do just that. When, for example, `max` is called with integer arguments, the compiler looks for a `max` function that can take two integers as argument. When that fails, the compiler then looks

```
template <typename T>
inline const T & max(const T & x, const T & y)
{
    return (x < y ? y : x);
}
```

Figure 6.4: A template for the `max` algorithm

```
// Prints the elements of v to cout, one per line,
// separated by a single space.
// Requirement on T: values can be printed to cout by
// using the output operator (<<).
template <typename T>
void println(const vector<T> & v)
{
    for (const T & e : v) cout << e << ' ';
    cout << '\n';
}
```

Figure 6.5: The generic function `println`

for a template that can be used to generate such a function. The template of Figure 6.4 will work if `T` is set to **`int`**. The process of generating a function out of a template is called **template instantiation**.[2]

Strictly speaking, a function template is not a function but we can think of it as a **generic function**, that is, a function that can work on more than one type of argument. The creation of generic functions 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.[3]

Another example of a generic function is shown in Figure 6.5. As documented there, it's a function that prints the contents of a vector on a single line. The documentation also clearly states a condition that the type `T` must meet. It is important to clearly identify such requirements when designing generic func-

[2]The template declaration of Figure 6.4 uses the keyword **`typename`** to declare the template parameter `T`. An alternative is to use the keyword **`class`**. This is allowed even if the type may not be a class. Although the keyword **`typename`** is more accurate, the use of the keyword **`class`** is widespread.

[3]In fact, Bjarne Stroustrup, the original designer of C++, describes the language as a better C that supports data abstraction, object-oriented programming and generic programming [Str].

```
template <class Iterator, class T>
int count(Iterator start, Iterator stop, const T & e)
{
    int count = 0;
    for (Iterator itr = start; itr != stop; ++itr) {
        if (*itr == e ) ++count;
    }
    return count;
}
```

Figure 6.6: An implementation of the generic algorithm `count`

tions. (In the case of the generic algorithm `max`, this was done for us by the C++ standard. See [CPP], for example.)

Figure 6.6 shows an implementation of the generic algorithm `count`. Note that this template has two parameters, one for the type of iterator and one for the type of element these iterators point to.

You may have noticed that in our implementation of `count`, the iterator arguments are passed by value. Up until now, we have been using the general rule that only small primitive values such as integers should be passed by value. Iterators often contain data that's no larger than a single integer, but not always. Still, it is generally better to pass iterators by value. The reason is that iterator arguments tend to be used repeatedly and accessing an iterator through a reference takes longer than accessing it directly. For example, in our implementation of `count`, if the number of elements in the range is $n$, then the iterator `stop` will be accessed $n+1$ times. It is more efficient to copy `stop` once when passing it as argument and then be able to directly access a copy of it. Therefore, as a general rule, we will always pass iterator arguments by value. The STL does the same.

The exercises of this section ask you to create your own implementations of various STL algorithms. To avoid possible conflicts with your compiler's imple-

```
namespace my {

... (your code)

} // namespace my
```

Figure 6.7: A namespace

mentations, and to allow you to choose which implementation you want to use in a particular piece of code, it is best to place your implementations in your own namespace, as shown in Figure 6.7. Then, to use your own implementation of `count`, for example, you would specify the namespace when calling the function (`my::count`) or you would include a **`using`** declaration in your code (**`using`** `my::count`).

Note that if the `algorithm` library is included in your code and you use `count` on a vector, for example, then the compiler will consider using `std::count` because it is in the same namespace as the class `vector`. This is a consequence of what is known as *argument-dependent lookup*. In that case, a declaration such as **`using`** `my::count` will cause the compiler to complain that the call to `count` is *ambiguous*. This simply means that the compiler doesn't know which version of `count` to use. In such a situation, you must specify the namespace when calling the function.[4]

## Study Questions

6.6.1. What is a generic function?

[4]As of March 2016, in the version of g++ that comes with the latest version of Code::Blocks, the `string` library includes the library file that contains the implementation of the algorithm `max`. This means that this ambiguity problem occurs with `max` and strings even if you don't include the `algorithm` library in your code.

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

6.6.3. What is generic programming?

### Exercises

6.6.4. Implement your own version of the following generic algorithms.

a) `swap`.

b) `max_element`.

c) `copy`.

d) `copy_backward`.

e) `reverse`. *Hint*: Make sure you consider sequences of even and odd length.

## 6.7 Initializer Lists

Table 6.7 includes a version of the `max` STL algorithm that takes an *initializer list* as argument:

```
max({elements})
```

We saw initializers lists earlier in these notes, as argument to one of the vector constructors:

```
vector<T> v({elements})
```

As we said back then, when such a list is specified by using braces, the elements should be separated by commas as in `{1, 2, 3}`.

In this section, we will learn how to implement the initializer-list version of `max`. In the process, we will learn the basics of how to program with initializer

```
template <typename T>
inline T max(initializer_list<T> init_list)
{
    return *max_element(init_list.begin(),
                        init_list.end());
}
```

Figure 6.8: An initializer-list version of the `max` algorithm

lists. We will use this knowledge again later when we learn how to implement our own class of vectors.

To create a version of `max` that takes an initializer list as argument, all we need to do is create a version of `max` that takes an object of class `initializer_list` as argument. A possible implementation is shown in Figure 6.8. This implementation uses the STL algorithm `max_element`.

The class `initializer_list` is generic. Therefore, just as in the case of `vector`, we need to specify the type of element stored in the initializer list. Initializer lists provide very limited functionality. In fact, they only support three methods: `size`, `begin` and `end`. Initializer-list iterators are random-access iterators.

You may have noticed that in our implementation of `max`, the initializer list argument is passed by value. Once again, this seems to violate the general rule that only small primitive values such as integers should be passed by value. But note that when an initializer list is copied, the copy is shallow: the elements are not copied, all that is copied is a reference to the elements. (Recall that we encountered the concept of a shallow copy in our discussion of arrays in Sect. 5.5.) This is in contract to the copying of vectors, which is always deep, in the sense that every element is copied. What this implies is that initializer lists can be copied quickly. And, as is the case with iterators, it is faster to directly access a copy of the list than to access the list through a reference. Therefore,

just like iterators, initializer lists are normally passed by copy. We will follow this rule and the STL does too.

The class `initializer_list` is defined in library file `initializer_list`, which is included in library file `utility`. The class is part of the `std` namespace.

### Study Questions

6.7.1. What three operations are supported by initializer lists?

### Exercises

6.7.2. Create a generic function `print(init_list, out)` that prints the elements of initializer list `init_list` to output stream `out`. The elements are printed on separate lines.

## 6.8 Functions as Arguments

Section 6.5 described several STL algorithms. But the STL includes more general versions of some of these algorithms. For example, we know that

```
count(start, stop, e)
```

counts the number of elements that are equal to `e`. But the more general

```
count_if(start, stop, condition)
```

counts the number of elements that meet the given `condition`. The `condition` argument should be a *unary predicate*, that is, a Boolean function that takes a single argument. The `count_if` algorithm applies this function to each of the elements in the range and counts how many of them cause the function to evaluate to **`true`**.

For example, suppose that we want to count the number of positive elements in a vector of integers. We would first create a function that tests if an integer is positive:

```
inline bool is_positive(int x) { return x > 0; }
```

Then, we would call `count_if` with that function:

```
count_if(v.begin(), v.end(), is_positive)
```

This will cause `is_positive` to be called on each element of `v`. The number of elements that satisfy `is_positive` (that is, that cause `is_positive` to return **`true`**) will be returned by `count_if`.

Now, how can this be implemented? How can functions be passed to other functions as arguments? In C++, the most basic way of passing functions as arguments is somewhat messy. A more general and cleaner approach is to use generic programming. For example, Figure 6.9 shows an implementation of the generic algorithm `count_if`. When this function is called with a function as its third argument, the compiler will set the template parameter `UnaryPredicate` to the appropriate type. We don't even need to know what that is.

Tables 6.8 and 6.9 list some generic algorithms that take functions as arguments. In all cases, these algorithms are generalizations of algorithms we saw in Section 6.5. Many other algorithms are described in a reference such as [CPP].

## Study Questions

6.8.1. What is one way of passing functions as arguments to another function?

## Exercises

6.8.2. Create a unary predicate `is_even` that determines if its integer argument is even. Then use it to count the number of even elements in some vector of integers.

`count_if(start, stop, condition)`
: Like `count` but only counts the number of elements that satisfy the unary predicate `condition`.

`find_if(start, stop, condition)`
: Like `find` but searches for the first element that satisfies the unary predicate `condition`.

`copy_if(start, stop, dest_begin, condition)`
: Like `copy` but only copies elements that satisfy the unary predicate `condition`.

`replace_if(start, stop, condition, y)`
: Like `replace` but replaces by `y` only the elements that satisfy the unary predicate `condition`.

`for_each(start, stop, f)`
: Calls the unary function `f` on each element in the range `[start,stop)`, in order. Returns the function.

Table 6.8: Some generic algorithms that take functions as arguments (part 1 of 2)

```
template <class Iterator, class UnaryPredicate>
int count_if(Iterator start, Iterator stop,
             UnaryPredicate condition)
{
    int count = 0;
    for (Iterator itr = start; itr != stop; ++itr) {
        if (condition(*itr)) ++count;
    }
    return count;
}
```

Figure 6.9: An implementation of the generic algorithm `count_if`

| |
|---|
| `max_element(start, stop, compare)` |
| `min_element(start, stop, compare)` |
| `binary_search(start, stop, e, compare)` |
| `lower_bound(start, stop, e, compare)` |
| `upper_bound(start, stop, e, compare)` |
| `sort(start, stop, compare)` |
| `max(x, y, compare)` |
| `min(x, y, compare)` |
| `max({elements}, compare)` |
| `min({elements}, compare)` |
| Like the earlier version but uses the binary predicate `compare` instead of the operator <. |

Table 6.9: Some generic algorithms that take functions as arguments (part 2 of 2)

6.8.3. Create a function `shorter_than` that takes two strings `s1` and `s2` as arguments and evaluates to **`true`** if the `s1` is shorter than `s2`, or if `s1` and `s2` are of the same length and `s1 < s2`. Then use the function to sort a vector of strings according to this ordering.

6.8.4. Implement your own version of the following generic algorithms.

a) `find_if`.

b) `replace_if`.

c) The version of `max_element` that takes a comparison function as argument.

d) `copy_if`.

e) `for_each`.

## 6.9 Function Objects

Continuing the example of the previous section, suppose that we now want to count the number of elements that are greater than some threshold `t` whose value is to be determined at run time. One idea is to define a unary predicate `is_large(x)` that evaluates to **`true`** is `x` is larger than `t`. But how is `is_large` going to have access to `t`? We can't pass `t` to `is_large` as an argument because the function needs to be a unary predicate. One solution would be for `t` to be a global variable. As we know, this is less than ideal.

A better solution is to create a class of objects that can each be initialized with their own value of `t` and then act as an `is_large` function for that particular value of `t`. Such a class is shown in Figure 6.10. Suppose that `is_large` is of class `IsLarge` and initialized with `t`, as in

```
IsLarge is_large(t);
```

```
class IsLarge
{
public:
    IsLarge(int t0) : t(t0) {}
    bool operator()(int x) { return x > t; }
private:
    int t;
};
```

Figure 6.10: A class of function objects

Then, because of the overloading of the function call operator (the operator `()`), `is_large` will act as a "greater than `t`" function. That is, `is_large(x)` will return **`true`** if `x > t`. Because these objects act as functions, we call them **function objects**.

To count the number of elements greater than `t`, all we now need to do is create an `IsLarge` function object initialized with `t` and call `count_if` with that function object:

```
count_if(v.begin(), v.end(), IsLarge(t))
```

This works for integers but it also works for any other type of value that can be compared by using the greater-than operator. So it makes sense to turn the class `IsLarge` into a template, as shown in Figure 6.11. Then, if `v` is a vector with elements of type `T` and `t` is a value of type `T`,

```
count_if(v.begin(), v.end(), IsLarge<T>(t))
```

will return the number of elements in `v` that are greater than `t`. This will work for any type `T` that supports the greater-than operator.

Note that when creating an object whose type is given by a class template, the template parameters must be specified explicitly: `IsLarge<T>(t)`. This

```
// Requirement on T: values can be compared by using
// the > operator.
template <class T>
class IsLarge
{
public:
    IsLarge(const T & t0) : t(t0) {}
    bool operator()(const T & x) { return x > t; }
private:
    T t;
};
```

Figure 6.11: A generic class of function objects

is in contrast with a call to a generic function such as `max(x, y)`. In that case, the compiler can usually figure out how to instantiate the template arguments from the type of arguments provided.

All the STL generic algorithms, including those listed in Tables 6.8 and 6.9, work with either functions or function objects as arguments. This is simply a consequence of the fact that generic programming allows us to pass functions and function objects as arguments in exactly the same way. For example, the function `count_if` shown in Figure 6.9 can accept as its third argument any type of value that behaves as a unary Boolean function, including plain functions as well as function objects.

The STL defines several classes of standard function objects. For example, function objects of class `greater` act as the greater-than operator. That is, if `compare` is of class `greater`, then `compare(x,y)` returns **`true`** if `x > y`. These functions objects can be used, for example, to sort a vector of integers in decreasing order:

```
sort(v.begin(), v.end(), greater<int>())
```

Note again that the type must be specified explicitly when creating the function object.

Another example is the class of function objects `multiplies`. These act as the binary multiplication operator. A sample application is in conjunction with the `accumulate` generic algorithm. The basic version

```
accumulate(start, stop, initial_value)
```

returns the result of adding all the elements of the range to `initial_value`. The more general version

```
accumulate(start, stop, initial_value, binary_op)
```

does the same thing but uses `binary_op` instead of the addition operator. For example,

```
accumulate(v.begin(), v.end(), 1, multiplies<int>())
```

returns the product of all the elements in vector of integers.

The standard function objects are defined in the library file `functional`. The `accumulate` generic algorithm is defined in the library file `numeric`. All are part of the `std` namespace.

## Study Questions

6.9.1. What is a function object?

## Exercises

6.9.2. Create a class of function objects `IsMultiple` that satisfies the following specification: when initialized with integer *m*, an object of this class behaves as a unary predicate that determines if its integer argument is a multiple of *m*. Then use one of these function objects to count the number of multiples of 3 in some vector of integers.

# Chapter 7

# Linked Lists

A linked list holds a sequence of elements all of the same type, just like a vector or an array. 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.

## 7.1 A Simple Text Editor

We will illustrate the usefulness of linked lists by designing and implementing a simple text editor. This editor will mostly be an extension of the file viewer we created in Chapter 5. The editor will allow the user not only to view the contents of a file but also to modify the contents of the file. More precisely, the user will be able to add, remove and replace entire lines of text. But the user won't be able to edit the contents of individual lines. For example, the user won't be able to insert a word directly in the middle of a line. The only way to accomplish this will be by replacing the entire line by a new one.

Here are more details on how the editor works. Like the file viewer, the edi-

```
co—op.txt
──────────────────────────────────────────────────────────────
  1 List for Co—op
  2
  3 bread
  4 yogurt
> 5 cumin
  6 black beans
  7 chick pea flour
  8 toothpaste
  9


──────────────────────────────────────────────────────────────
  next      jump     insert  open  quit
  previous  replace  delete  save
────────
choice: i
new line: ginger
```

Figure 7.1: Sample user interface of the text editor

tor has a *buffer* that contains lines of text, usually an edited copy of the contents of some file. The editor displays the name of the file, if any, followed by a certain number of lines from the buffer, surrounded by a border. A cursor indicates the position of the current line. Below the text, a menu of commands is displayed followed by the prompt "`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 7.1 shows what the user interface looks like. The available editor commands are described in Figure 7.2.

Note that the displayed contents of the buffer always includes an extra empty

| | |
|---|---|
| `next` | The next line becomes the current line. |
| `previous` | The previous line becomes the current line. |
| `jump` | Asks for a line number and makes that line become the current line. |
| `replace` | Asks for a new line and replaces the current line. |
| `insert` | Asks for a new line and inserts it before the current line. |
| `delete` | Deletes the current line. |
| `open` | Asks for a file name and reads that file into the buffer. |
| `save` | Asks for a file name and saves the contents of the buffer to that file. |
| `quit` | Stops the editor. |

Figure 7.2: The commands of the text editor

line we will call the *end line*. In Figure 7.1, the end line is the one numbered 9. That line is not really part of the buffer but the cursor can move there. This allows the user to insert a new line at the end of the buffer. In addition, in the case of an empty buffer, the end line gives the cursor something to point to.

In this chapter, we will create two versions of the text editor. The only difference between these two versions will be the way in which the contents of the buffer is stored. The first version will use a vector. We will then discuss if this is really the best choice and discover an alternative: the linked list.

## 7.2 Vector Version of the Text Editor

The general behavior of the text editor was described in the previous section 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? One possible specification is shown in Figures 7.3 to 7.7. It includes answers to these questions as well as several other details.

A vector version of the text editor can be created by expanding the file viewer program we created in Chapter 5. The design of the program will be similar. A `TextEditor` component will perform the overall control of the program as well as most of the user interaction. A `Buffer` component will handle the storage of the buffer and the execution of the commands.

The text editor can be implemented by extending the implementation of the file viewer. In this section, we will highlight the new code. Figure 7.8 shows the declaration of the `Buffer` class. In addition to keeping track of the index of the top line, the `Buffer` also keeps track of the index of the current line. Both indices are initialized to 0. Figures 7.9 and 7.10 show the implementation of the `Buffer` methods that are new to the text editor.

The complete source code and documentation of this version of the text editor are available on the course web site under `TextEditor1.0`.

```
OVERVIEW

A simple text editor that allows the user to add,
remove and replace entire lines of text. It doesn't
allow the user to edit the contents of the lines. (The
only way to accomplish this is to replace the entire
line by a new one.)


DETAILS

The editor has a buffer that contains lines of text,
usually an edited copy of the contents of some file.
The editor interacts with the user as shown in the
following example:
```

Figure 7.3: Specification of the text editor (part 1 of 5)

```
co—op.txt
_____________________________________________________________

  1 List for Co—op
  2
  3 bread
  4 yogurt
> 5 cumin
  6 black beans
  7 chick pea flour
  8 toothpaste
  9


_____________________________________________________________

  next      jump     insert  open  quit
  previous  replace  delete  save
_______

choice: i
new line: ginger

The program begins by asking the user for a window
height. This is the maximum number of lines that will
be displayed at any time. The displayed lines are
numbered starting at 1 for the first line of the file.
If the number of lines on the last page is smaller
than the window height, the rest of the window is
filled with unnumbered empty lines.
```

Figure 7.4: Specification of the text editor (part 2 of 5)

```
Buffer lines are displayed between two lines of 50
dashes. The name of the file is printed above the
first line of dashes. A cursor (>) indicates the
position of the current line. Below the second line of
dashes, a menu of commands is displayed. Below that
menu, the prompt "choice:" is displayed. The user
types the first letter of a command, the command
executes and everything is redisplayed. Some commands
prompt the user for more information.

The displayed contents of the buffer always includes
an extra empty line that we 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. This is also the
line that the crusor points to when the buffer is
empty.

Here is a description of the various commands:

next: The next line becomes the current line. Does
nothing if the current line is the end line.

previous: The previous line becomes the current line.
Does nothing if the current line is the first one.
```

Figure 7.5: Specification of the text editor (part 3 of 5)

```
jump: Asks for a line number (with prompt "line
number:") and makes that line become the current line.
The new current line is displayed at the top. If the
user enters an invalid line number N, the message
"ERROR: N is not a valid line number" is displayed
just before the file name is redisplayed.

replace: Asks for a new line (with prompt "new line:")
and replaces the current line.

insert: Asks for a new line (with prompt "new line:")
and inserts it before the current line. The line that
follows the new one becomes the current one. (This
makes it easy to add a sequence of new lines, in
order.)

delete: Deletes the current line.  The next line
becomes the current one. Nothing happens if the user
tries to delete the end line.

open: Asks for a file name (with prompt "file name:")
and reads that file into the buffer. If a file named X
does not open, the message "ERROR: Could not open X"
is displayed just before the file name is redisplayed.

save: Asks for a file name (with prompt "file name:")
and saves the contents of the buffer to that file. If
a file named X does not open, the message "ERROR:
Could not open X" is displayed just before the file
name is redisplayed.
```

Figure 7.6: Specification of the text editor (part 4 of 5)

```
quit: Stops the editor.


NOTES FOR LATER VERSIONS

open and quit should check if the buffer has been
saved.

Add more error—checking. (Check that commands are
entered properly and that the window height is a
positive integer.)
```

Figure 7.7: Specification of the text editor (part 4 of 5)

## Exercises

7.2.1. 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. Refer to the program specification for details on this command's behavior.

c) Modify the *save* command so it uses the current file, if any, as a default value. If the user enters an empty file name, then the default is used.

d) Add a *clear* command that empties the buffer. The command also resets the file name so that the string `<no file opened>` is displayed just like when the program is started.

e) Add *Next* and *Previous* commands that cause the editor to display the next or previous "pages", just like the *next* and *previous* commands of the file viewer. After these commands are executed, the new top line becomes the current line.

```
class Buffer
{
public:
    void display() const;
    void erase();
    const std::string & get_file_name() const {
        return file_name;
    }
    void insert(const std::string & new_line);
    void move_to_next_line();
    void move_to_previous_line();
    bool open(const std::string & new_file_name);
    bool save(const std::string & new_file_name);
    void set_window_height(int h) {
        window_height = h;
    }

private:
    std::vector<std::string> v_lines;
    int ix_current_line = 0;
    int ix_top_line = 0;
    std::string file_name;
    int window_height;
};
```

Figure 7.8: Declaration of `Buffer`

```
inline void Buffer::erase()
{
    if (ix_current_line < v_lines.size())
        v_lines.erase(v_lines.begin() +
                          ix_current_line);
}

inline void Buffer::insert(
        const std::string & new_line)
{
    v_lines.insert(v_lines.begin() +
                       ix_current_line, new_line);
    move_to_next_line();
}

inline void Buffer::move_to_next_line()
{
    if (ix_current_line < v_lines.size()) {
        ++ix_current_line;
        // check if window needs to be scrolled down
        if (ix_current_line >=
                ix_top_line + window_height)
            ++ix_top_line;
    }
}
```

Figure 7.9: Implementation of some of the `Buffer` methods (part 1 of 2)

```
inline void Buffer::move_to_previous_line()
{
    if (ix_current_line > 0) {
        --ix_current_line;
        // check if window needs to be scrolled up
        if (ix_current_line < ix_top_line)
            --ix_top_line;
    }
}

bool Buffer::save(const string & new_file_name)
{
    std::ofstream file(new_file_name);
    if (!file) return false;

    file_name = new_file_name;
    for (const string & s : v_lines) file << s << '\n';
    return true;
}
```

Figure 7.10: Implementation of some of the `Buffer` methods (part 2 of 2)

f) Add an `Append` command that inserts a new line *after* the current one. Make sure you consider special cases.

g) The *quit* command currently stops the editor without ensuring that the buffer was saved to a file. Similarly, the *open* and *clear* commands delete the current contents of the buffer without ensuring that this contents has been saved. Fix this. In case the buffer has not been saved since it was last modified, *clear*, *open* and *quit* should ask the user if he or she wants to save the current contents of the buffer. *Hint*: Add a Boolean variable to the buffer that indicates if the buffer was saved since the last modification.

h) Add a *search* command that asks the user for a string and finds the first line that contains that string. The search starts at the current line and stops at the end of the file. If the string is found, the line that contains it becomes the current line and is displayed at the top. In case the user enters a string *X* that does not occur in the file, the program should print the error message *ERROR: string "X" was not found*.

i) Modify the *search* command so that the search wraps around to the beginning of the file when the end of the file is reached.

j) Add an *again* command that repeats the last search. If that last search was the previous command, then the new search starts at the second line that is currently displayed. If no search has been previously performed, then the *again* command asks the user for a string.

7.2.2. Add more error checking to the file viewer. Change the program as little as possible.

a) Make the program check that the user enters a positive integer as the window height. If not, the program should print an error message and ask again.

b) Make the program check that the user enters either the whole name of a command or the first letter of the command. If the user enters an invalid command *X*, the program should print the error message *ERROR: X is not a valid command*. The error message should be displayed at the top of the window, just like when a file does not open.

## 7.3 Vectors and Linked Lists

In the previous section, we used a vector to store the contents of the buffer in the text editor. 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, the *next* command runs in constant time since, as we saw in the previous section, all it does is essentially add 1 to the index of the current line.

Other commands, however, require much more work. The *insert* command uses the vector method `insert` but, in the worst case, that method runs in linear time. Here's why. The vector indexing operator runs in constant time. This is normally achieved by storing the elements of a vector side-by-side in the computer's memory. But a consequence of this is that inserting a new element in the middle of the vector requires moving, one position to the right, all the elements of the vector, from the insertion point to the end.

Table 7.1 lists all the commands of the text editor. The second column of Table 7.1 shows the running times of all the editor commands when 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 *next* and *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?

| | *vector* | *linked list* |
|---|---|---|
| `next` | $\Theta(1)$ | $\Theta(1)$ |
| `previous` | $\Theta(1)$ | $\Theta(1)$ |
| `jump` | $\Theta(1)$ | $\Theta(n)$ |
| `replace` | $\Theta(1)$ | $\Theta(1)$ |
| `insert` | $\Theta(n)$ | $\Theta(1)$ |
| `delete` | $\Theta(n)$ | $\Theta(1)$ |
| `open` | $\Theta(n)$ | $\Theta(n)$ |
| `save` | $\Theta(n)$ | $\Theta(n)$ |
| `quit` | $\Theta(1)$ | $\Theta(1)$ |

Table 7.1: Running time of the editor commands

Since the linear running time of these methods is caused by the fact the buffer lines are stored side-by-side in the computer's memory, an idea for a new data structure would therefore be not to store buffer lines 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 the *linked list*. Later in these notes, we will learn how to implement linked lists. In this chapter, we will learn to use the STL container `list`, which is a class of linked lists.

The main advantage of linked lists is that inserting or deleting an element at a given position can be done quickly, as long as we have a direct link to the element at that position. 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 7.1 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 or fast insertions and deletions, but not both. 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, a linked list is probably a better choice for a text editor.

## Study Questions

7.3.1. What is the key difference between vectors and linked lists?

## 7.4 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 7.2 to 7.4 show some of the most basic list operations. (The methods `begin`, `end` and their usual variations are not included.) The running time of the operations is also indicated. Unless otherwise specified, the parameter $n$ is the initial number of elements held by the receiver.

Note that STL lists don't provide an indexing operator or any other method that gives access to an element given its index. This is because such an operation would be inefficient as it would require the list to be traversed from the beginning. Given an index `i`, it's still possible to set an iterator to point to the element with index `i`:

```
itr = ls.begin();
advance(itr, i);
```

But this code runs in time linear in the value of `i` and lists don't provide the convenience of an indexing operator: in situations where such an operation is frequently needed, a vector should probably be used instead.

Note that list iterators are bidirectional iterators. Recall that this means that they support the operations `*`, `->`, `++`, `--`, `==` and `!=`, but not the random-access operations `+i`, `-i`, `-`, `<` and `[i]`.

The `list` container is defined in library file `list` and included in the `std` namespace. STL lists include several additional operations that are described in a reference such as [CPP].

### Study Questions

7.4.1. What mechanism is used to specify positions within an STL list?

`list<T> ls`
`list<T> ls(n)`
`list<T> ls(n, e)`
`list<T> ls({elements})`
`list<T> ls(start, stop)`
`list<T> ls(ls2)`
: Creates a list `ls` that can hold elements of type `T`. The list is initialized to be empty, or to contain `n` value-initialized elements of type `T`, or `n` copies of element `e`, or copies of the given `elements`, or copies of all the elements in the range [`start`, `stop`), or copies of all the elements of list `ls2`. $\Theta(1)$ for the first constructor, $\Theta(n)$ for the others. In the case of the last three constructors, $n$ is the size of the argument.

`ls.size()`
: Asks list `ls` for the number of elements it currently contains. $\Theta(1)$.

`ls.empty()`
: Asks list `ls` if it is empty. $\Theta(1)$.

`ls.front()`
`ls.back()`
: Asks list `ls` for a reference to its front or back element. $\Theta(1)$.

`ls.push_front(e)`
`ls.push_back(e)`
: Asks list `ls` to add a copy of element `e` to its front or back. $\Theta(1)$.

`ls.pop_front()`
`ls.pop_back()`
: Asks list `ls` to delete its first or last element. $\Theta(1)$.

Table 7.2: Some list operations (part 1 of 2)

`ls1 = ls2`
`ls1 = {elements}`
: Makes list `ls1` contain copies of all the elements of list `ls2`, or copies of the given `elements`. Returns a reference to `ls1`. $\Theta(n)$, where $n$ is the total size of `ls1` and `ls2`.

`ls1.swap(ls2)`
: Asks list `ls1` to swap contents with list `ls2`. $\Theta(1)$.

`ls.assign(n, e)`
`ls.assign({elements})`
`ls.assign(start, stop)`
: Asks list `ls` to change its contents to `n` copies of element `e`, or to copies of the given `elements`, or to a copy of all the elements in the range [`start`, `stop`). $\Theta(m)$, where $m$ is the greater of the initial size and the new size of `ls`.

`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`. $\Theta(k)$, where $k$ is the number of elements deleted or inserted.

Table 7.3: Some list operations (part 2 of 3)

`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, {elements})`
`ls.insert(itr, start, stop)`
: Asks list `ls` to insert, at the position indicated by the iterator `itr`, copies of the given `elements`, or copies of all the elements in the range [`start`, `stop`). An iterator that points to the first new element is returned. $\Theta(k)$, 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`). An iterator that points to the next element is returned. $\Theta(k)$, where $k$ is the number of elements deleted.

`ls.remove(e)`
`ls.remove_if(condition)`
: Asks list `ls` to delete all elements that equal `e` or that satisfies the unary predicate `condition`. $\Theta(n)$.

`ls.clear()`
: Asks list `ls` to delete all its elements. $\Theta(n)$.

Table 7.4: Some list operations (part 3 of 3)

### Exercises

7.4.2. Experiment with lists by writing a test driver that creates more than one type of list and uses all the methods shown in Tables 7.2 to 7.3.

7.4.3. Create a generic function `print(ls)` that prints list `ls` to standard output (`cout`). Elements are printed on separate lines by using the output operator (`<<`).

7.4.4. Create a generic function `concatenate(ls1, ls2, result)` that makes list `result` contain a copy of all the elements of list `ls1` followed by a copy of all the elements of list `ls2`.

7.4.5. Create a function `make_double_spaced(ls)` that takes as argument a list of strings and inserts a new empty string after every string in the list.

7.4.6. Create a generic function `insert(ls, itr, start, stop)` that behaves exactly like `ls.insert(itr, start, stop)`. (Don't use that version of `insert` in implementing your function, but you can use the `insert(itr, e)` version.)

7.4.7. Create a generic function `erase(ls, start, stop)` that behaves exactly like `ls.erase(start, stop)`. (Don't use that version of `erase` in implementing your function, but you can use the `erase(itr)` version.)

## 7.5 List Version of the Text Editor

Because STL vectors and lists have a very similar interface (methods, operators and constructors), it is fairly easy to transform the vector version of our text editor so it uses a list instead.

One difference is that in addition to keeping track of the indices of the current and top lines, `Buffer` also needs to maintain iterators that point to those lines.

```
inline void Buffer::erase()
{
    if (ix_current_line < ls_lines.size()) {
        itr_current_line =
            ls_lines.erase(itr_current_line);
        if (ix_top_line == ix_current_line)
            itr_top_line = itr_current_line;
    }
}

inline void Buffer::insert(
        const std::string & new_line)
{
    itr_current_line =
        ls_lines.insert(itr_current_line, new_line);
    if (ix_top_line == ix_current_line)
        itr_top_line = itr_current_line;
    move_to_next_line();
}
```

Figure 7.11: The `Buffer` methods `erase` and `insert`

This is to be able to efficiently display the buffer and perform operations at the location of the current line.

Another significant difference is in the implementation of the `Buffer` methods `erase` and `insert`. Both are shown in Figure 7.11. In the `erase` method, after the deletion, the iterator to the current line needs to be set. In addition, in case the current line was also the top line, the iterator to the top line also needs to be set. In the case of `insert`, the iterator to the current line needs to be set to point to the new line. The iterator to the top line may also need to be adjusted.

The complete source code and documentation for the list version of the text editor is available on the course web site under `TextEditor2.0`.

## Exercises

7.5.1. Repeat Exercise 7.2.1 on the list version of the text editor.

7.5.2. Repeat Exercise 7.2.2 on the list version of the text editor.

# Chapter 8

# 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.

## 8.1 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 8.1 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 of a command, the command executes and the appropriate entry is displayed. Some commands prompt the user for more information. The available phone book commands are described in Figure 8.2. 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

```
John Smith
123-456-7890
_____________________________________________________________

  next       search  edit    quit
  previous   add     delete
_________

command: e
new number:
```

Figure 8.1: Sample user interface of the phone book program

| | |
|---|---|
| `next` | The next entry is displayed. Wraps around. |
| `previous` | The previous entry is displayed. Wraps around. |
| `search` | Asks for a name and displays the corresponding entry. If not found, the earlier entry is redisplayed. |
| `add` | Asks for a new name and phone number, adds a new entry to the phone book and displays the new entry. |
| `edit` | Asks for a new number and edits the current entry. Does nothing if the phone book is empty. |
| `delete` | Deletes the current entry. Displays the next one (with possible wrap-around). Does nothing if the phone book is empty. |
| `quit` | Saves the phone book to a file and halts the program. |

Figure 8.2: The commands of the phone book

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 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

| | search | add | delete |
|---|---|---|---|
| Sorted vector plus binary search | $\Theta(\log n)$ | $\Theta(n)$ | $\Theta(n)$ |
| Balanced binary search tree | $\Theta(\log n)$ | $\Theta(\log n)$ | $\Theta(\log n)$ |
| Hash table | $\Theta(1)$ | $\Theta(1)$ | $\Theta(1)$ |

Table 8.1: Summary of the various options discussed in this section

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 8.1 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 guarantees the performance of balanced binary search trees and is typically implemented using that data structure, as well as a container `unordered_map` that is a class of hash tables.

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.

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.

Note that there are some circumstances where a sorted vector is preferable to a map. One 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

[1] At Clarkson, these data structures are covered in detail in the course CS344 *Algorithms and Data Structures*.

more memory than the optimal amount of a sorted vector of just the right size.

Another circumstance would be if 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.

In general, however, with indexed data, maps are usually the best choice. Situations where sorted vectors are preferable are fairly specialized and not that common.

## 8.2 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.

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 balanced 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 be 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`.

Tables 8.2 to 8.4 show some additional map operations. (The methods `begin`, `end` and their usual variations are not included.) The running times of the operations 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.

Map iterators are bidirectional iterators. But 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.

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 balanced 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

`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[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.count(k)`
: Asks map `m` for the number of elements it contains that have key `k`. Either 0 or 1 is returned. $\Theta(\log n)$.

`m1 = m2`
: Makes map `m1` a copy of map `m2`. Returns a reference to `ls1`. $\Theta(n)$, 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 8.2: Some map operations (part 1 of 3)

`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. An iterator that points to the next pair is returned. Amortized $\Theta(1)$.

`m.erase(start, stop)`
: Asks map `m` to delete all elements in the range [`start`,`stop`). An iterator that points to the next pair is returned. $\Theta(\log n + r)$, where $r$ is the number of elements removed.

`m.clear()`
: Asks map `m` to delete all its elements. $\Theta(n)$.

Table 8.3: Some map operations (part 2 of 3)

`m.insert(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 a pair consisting of an iterator and a Boolean value. If the insertion is successful, the Boolean value is **`true`** and the iterator points to the new pair. If the insertion fails, the Boolean value is **`false`** and the iterator points to the pair that prevented the insertion. $\Theta(\log n)$.

`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 or to the pair that prevented the insertion. The iterator `itr` is an indication of where the new pair may belong. Amortized $\Theta(1)$ if the new pair is inserted right before the position indicated by `itr`. Otherwise, $\Theta(\log n)$.

Table 8.4: Some map operations (part 3 of 3)

key.

The second `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 as [CPP].

## Study Questions

8.2.1. What is a map?

8.2.2. How many types need to be specified when creating a map? What do they represent?

8.2.3. What happens when the indexing operator is used with a key that is not present in a map?

8.2.4. What type of value do map iterators point to?

## Exercises

8.2.5. Experiment with maps by writing a test driver that creates more than one type of map and uses all the methods shown in Tables 8.2 and 8.3.

8.2.6. 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.

8.2.7. 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.

## 8.3 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 implementation of the program.

As decided earlier, the program will have a class of phone book entries that each holds a name and a phone number. These objects are responsible for their own reading and printing (just like `Time` and `string` objects).

The implementation of this class is shown in Figure 8.3. 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 rest of the phone book program can be designed in a way similar to the text editor of the previous chapter. The storage of the phone book entries and

```
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 8.3: The declaration and implementation of `PhoneBookEntry`

```
class PhoneBook
{
public:
    void run();

private:
    void display_entry_and_menu() const;
    void execute(char command, bool & done);

    PhoneBookList entry_list;
};
```

Figure 8.4: The declaration of the `PhoneBook` class

the execution of the phone book commands on those entries will be handled by a `PhoneBookList` component. The overall control of the phone book program, and most of the user interaction, will be handled by a `PhoneBook` component.

Figure 8.4 shows the declaration of the `PhoneBook` class. This class has a public method `run` as well as two private helper methods, `display_entry_and_menu` and `execute`. Figures 8.5 to 8.7 show the implementations of `run` and `execute`.

Figure 8.8 shows the declaration of the `PhoneBookList` class. The iterator `itr_current_entry` marks the location of the current line.

Figures 8.9 and 8.10 show the implementation of most of the `PhoneBookList` methods. The `add` method uses the map `insert` method, which conveniently returns an iterator to the new entry, or to an existing entry that already has the given name. Note how, in the arguments of `insert`, we don't have to explicitly construct first a `PhoneBookEntry` and then a `pair` of the right type:

```
void PhoneBook::run()
{
    entry_list.read_file(kcsFileName);
    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 8.5: The `run` method

```
    m_entries.insert(
        std::pair<std::string, PhoneBookEntry>(
            name, PhoneBookEntry(name, number)));
```

These objects are constructed automatically from the initializer lists provided as arguments to `insert`:[2]

```
    m_entries.insert({name, {name, number}});
```

Figure 8.11 shows the implementation of the `read_file` method. After opening the file and reading the number of entries, each entry is read and added to the map by using the version of `insert` that takes an iterator as argument.

It would have been slightly simpler to add each new entry to the map by using the indexing operator:

[2]The ability to construct objects automatically from initializer lists is new to C++11.

```
void PhoneBook::execute(char command, bool & done)
{
    switch (command) {
        case 'n': {
            entry_list.move_to_next();
            break;
        }
        case 'e': {
            if (entry_list.empty()) return;
            cout << "new number: ";
            string new_number;
            getline(cin, new_number);
            entry_list.edit_current(new_number);
            break;
        }
        case 's': {
            cout << "name: ";
            string name;
            getline(cin, name);
            entry_list.find(name);
            break;
        }
        ...
    };
}
```

Figure 8.6: The `execute` method (part 1 of 2)

```
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);
            entry_list.add(new_name, new_number);
            break;
        }
        case 'q': {
            entry_list.write_file(kcsFileName);
            done = true;
            break;
        }
    };
}
```

Figure 8.7: The `execute` method (part 2 of 2)

```
class PhoneBookList
{
public:
    PhoneBookList() :
        itr_current_entry(m_entries.end()) {}

    void add(const std::string & name,
             const std::string & number);
    void display_current_entry() const;
    void move_to_next();
    void edit_current(const std::string & new_number);
    void find(const std::string & name);
    bool empty() const { return m_entries.empty(); }
    void read_file(const std::string & file_name);
    void write_file(const std::string & file_name)
        const;

private:
    std::map<std::string, PhoneBookEntry> m_entries;
    std::map<std::string, PhoneBookEntry>::iterator
        itr_current_entry;
};
```

Figure 8.8: Declaration of `PhoneBookList`

```
inline void PhoneBookList::add(
        const std::string & name,
        const std::string & number)
{
    auto result =
        m_entries.insert({name, {name, number}});
    itr_current_entry = result.first;
}

inline void PhoneBookList::display_current_entry()
        const
{
    if (m_entries.empty()) return;
    std::cout << itr_current_entry->second;
}

inline void PhoneBookList::move_to_next()
{
    if (m_entries.empty()) return;
    ++itr_current_entry;
    if (itr_current_entry == m_entries.end()) {
        itr_current_entry = m_entries.begin();
    }
}
```

Figure 8.9: Some of the `PhoneBookList` methods (part 1 of 2)

```
inline void PhoneBookList::edit_current(
        const std::string & new_number)
{
    itr_current_entry->second.number = new_number;
}

inline void PhoneBookList::find(
        const std::string & name)
{
    auto itr = m_entries.find(name);
    if (itr != m_entries.end())
        itr_current_entry = itr;
}
```

Figure 8.10: Some of the `PhoneBookList` methods (part 2 of 2)

```
    m_entries[new_entry.name] = new_entry;
```

But this would not have been as efficient. Here's why. 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 $\Theta(\log(n/2)) = \Theta(\log n - 1)$, which is $\Theta(\log n)$. This leads to a total reading time that's at least $\Theta(n \log n)$. (The running time is also at most $\Theta(n \log n)$ because each indexing operation takes time at most $\Theta(\log n)$.)

In contrast, if we make sure that the entries in the file are always sorted in increasing order by name then, as we read the entries from the file, we know where each new entry needs to be inserted: at the end of the map. By using the `insert` method with an end iterator, each entry is added in amortized constant time, for a total reading time of $\Theta(n)$.

The source code and documentation of the phone book program are available on the course web site under `PhoneBook1.0`. Note that the implementation of the program is not complete; one of the exercises asks you to add what's missing.

```
void PhoneBookList::read_file(
        const std::string & file_name)
{
    ifstream ifs(file_name);
    if (!ifs) return;
        // no file; one will be created when
        // write_file is called

    int num_entries;
    ifs >> num_entries;
    ifs.get(); // \n
    for (int i = 0; i < num_entries; i++) {
        PhoneBookEntry new_entry;
        ifs >> new_entry;
        m_entries.insert(m_entries.end(),
                         {new_entry.name, new_entry});
    }
    itr_current_entry = m_entries.begin();
}
```

Figure 8.11: The `read_file` method

## Exercises

8.3.1. 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.

8.3.2. 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 9

# Object-Oriented Design

We have already created four programs in these notes: a pay calculator, a file viewer, 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, vectors, linked lists and maps. In this chapter, we will step back and discuss the software development process in more detail.

## 9.1 The Software Life Cycle

As we saw when we created the pay calculator, text editor and phone book programs earlier in these notes, each new piece of software needs to be specified, designed and implemented.

**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.

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 the user code (or client code) uses to communicate with the component. 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 software maintenance, for short.

Software specification, design, implementation and maintenance are often referred to as the four stages of the **software life cycle**. We will say a bit more about the first three stages later in this chapter. But first, in the next section, we discuss a key issue in the management of the software development process.

### Study Questions

9.1.1. What are the four stages of software development?

9.1.2. What is the interface of a software component?

9.1.3. What is integration testing?

9.1.4. What is the main advantage of integration testing (over testing all the components together right away)?

## 9.2 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, for example, with the pay calculator: we built a first version that performed no error-checking.

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, Larman and 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."

In addition, Fred Brooks, an early leader in the field of software engineering,[1] described the essential challenges of software engineering and identified incremental development as one of three promising solutions [Bro87]. 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 *agile* approaches to software development.

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*.

## Study Questions

9.2.1. What is a disadvantage of coding right away without first specifying and designing the software?

9.2.2. What is the waterfall model of software development?

9.2.3. What is incremental software development?

9.2.4. What are the benefits of incremental development?

[1] Brooks was also the 1999 winner of the ACM's Turing Award, which is widely regarded as the "Nobel Prize" of computer science [TA].

## 9.3 Specification, Design and Implementation

In this section, we say a bit more about the specification, design and implementation of a program.

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*. This allows us to explore and specify every detail of the program's behavior.

Recall that our goal is to design a modular program. There are three strategies that help us achieve this goal. First, of course, we 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. 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.

Brooks 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 "see" our programs.

Figure 9.1 shows the components of the phone book program we created in the previous chapter. Recall from earlier in these notes that in a component diagram, each component is represented by a box and an arrow from one com-

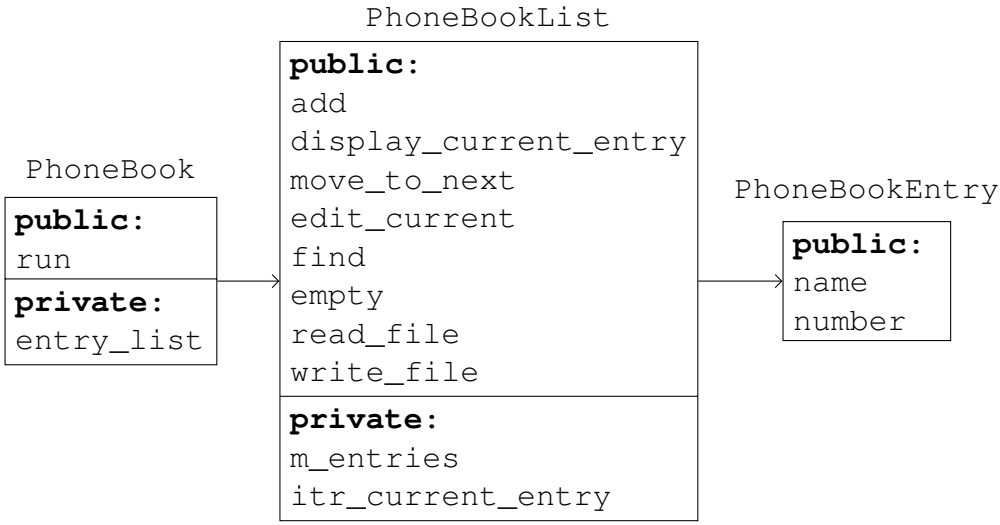


Figure 9.1: A high-level component diagram

ponent to another means that the first component uses the second one. The main methods and data members of each component are indicated. A horizontal line separates the public members from the private ones.

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 9.2 shows an *interaction diagram*. In an interaction diagram, each type of component is represented by a vertical line and a labeled arrow represents a message being sent from an object of one class to an object of another class. Unlabeled arrows represent control returning to the first object. A component diagram is like a photograph of the program; an interaction diagram is more like a video.

In the diagram of Figure 9.2, we have chosen to include the map component held by the `PhoneBookList`.

This particular interaction diagram illustrates the launching of the program

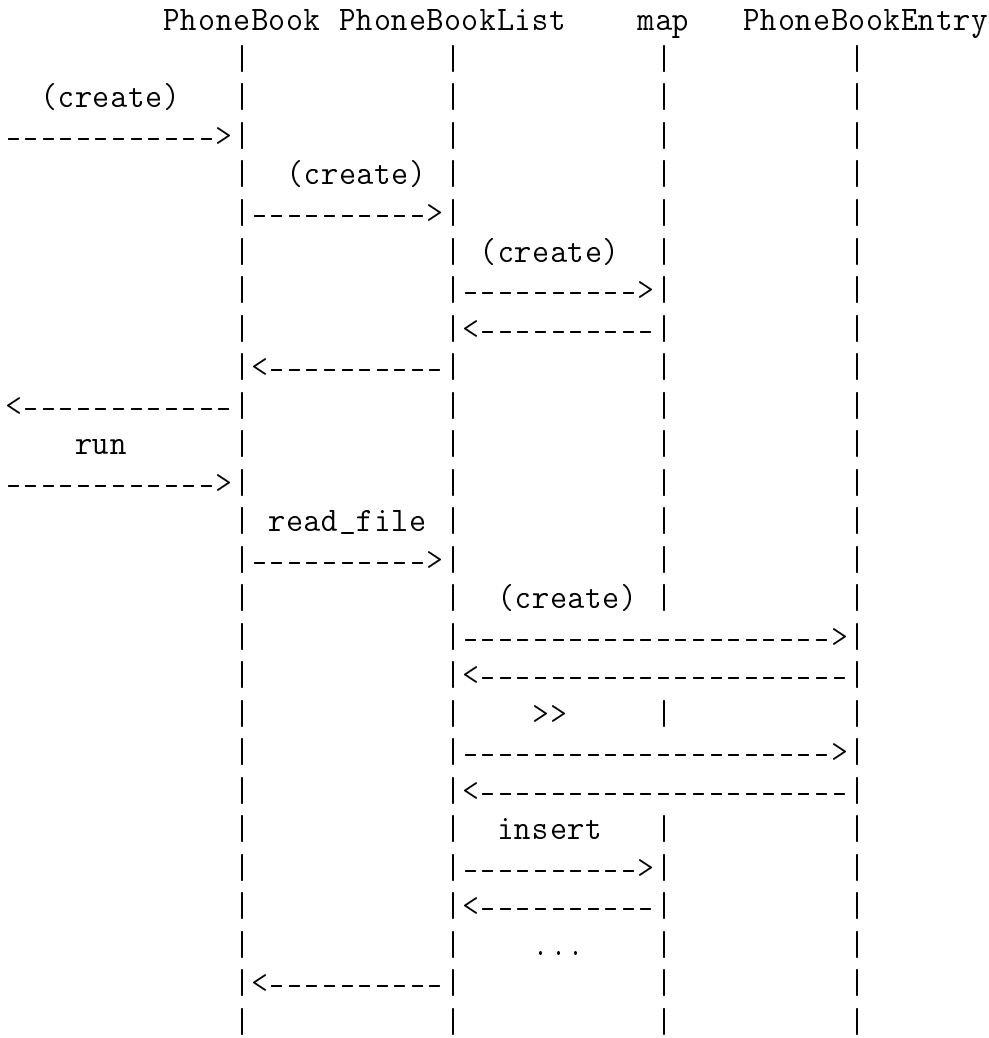


Figure 9.2: Launch of the phone book program up to the reading of the file

up to the reading of the file. A single interaction diagram can describe the entire execution of only a very small program. For larger programs, such as our phone book program, multiple diagrams are needed to describe all the possible scenarios. For example, Figure 9.3 illustrates the following scenario: the user searches for a an entry and then edits it. That interaction diagram does not show the details of the displaying of the current entry. Those are shown in Figure 9.4.

As we said earlier, after coding, each component should be tested on its own before it is integrated into the rest of the program. But if a component uses other components, then it cannot be tested in complete isolation. One option is to wait until the other components have been implemented. In a single-person project, this may work well. But in a multi-person project, we'd like to have all the developers coding and unit testing their components at the same time.

So another option is to create **stubs** for the other components. A stub is a dummy version of a component: it has the same interface but performs none or only a very small part of the intended responsibilities of the component. For example, in the phone book program, we could test the `PhoneBook` class, which is mainly responsible for interacting with the user, by using a dummy `PhoneBookList` class that does nothing except always display the same fake entry. (This requires the creation of stubs for each of the `PhoneBookList` methods.) Testing `PhoneBook` with a dummy `PhoneBookList` would allow us to verify that `PhoneBook` performs the user interaction properly. Of course, this doesn't tell us if `PhoneBook` will work properly with the real `PhoneBookList`. This is to be determined at integration testing, after `PhoneBookList` has been implemented (and tested on its own).

## Study Questions

9.3.1. What does working through scenarios mean?

9.3.2. Why should we aim for a design in which each component has its own secret?

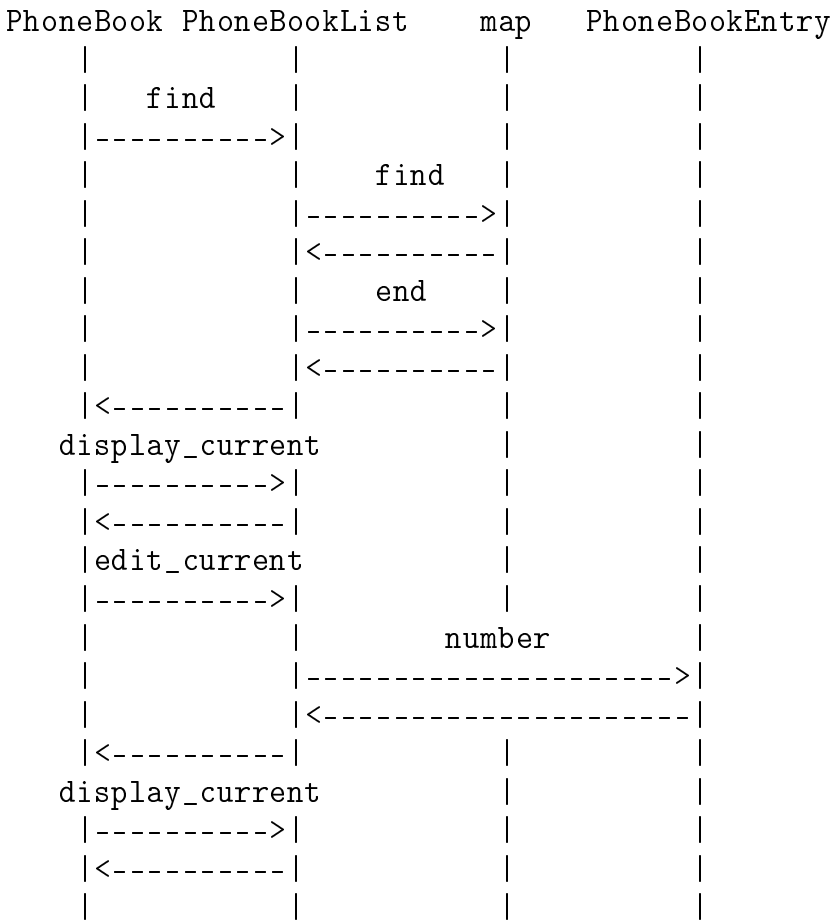


Figure 9.3: The user searches for a an entry and then edits it

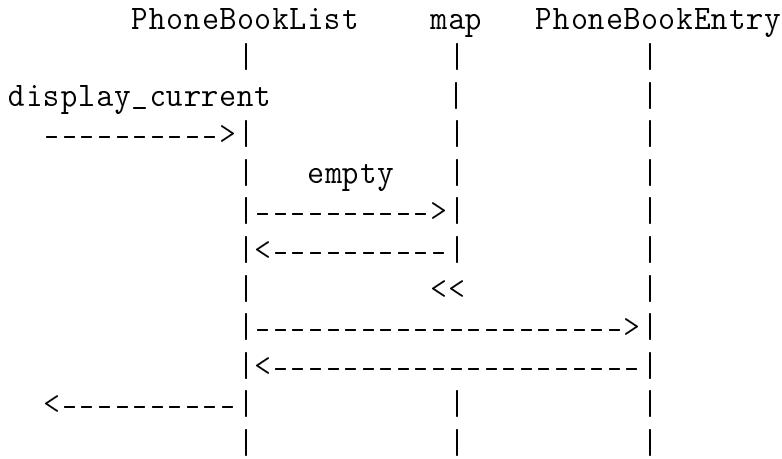


Figure 9.4: The displaying of the current entry

9.3.3. What is the what-who cycle?

9.3.4. What is responsibility-driven design?

9.3.5. What are two benefits of including in the design of the phone book program a component for individual entries?

9.3.6. What is a component diagram?

9.3.7. What is an interaction diagram?

9.3.8. What is a stub or dummy component?

# Chapter 10

# Dynamically Allocated Arrays

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.

## 10.1 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.[1]

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 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

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

## 10.2 The Dynamic Allocation of Arrays

In previous section, we learned that for an array to have a size that's determined at run time, the array must be stored outside the activation record of a function,

[1] At Clarkson, courses such as CS241 *Computer Organization* and CS445 *Compiler Construction* normally cover this subject in more detail.

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];
```

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,

```
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 10.1 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.

```
int n;
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 10.1: Creating and accessing a dynamically allocated array

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

10.2.1. What does the **`new`** operator do?

10.2.2. When allocating an array, what value does the **`new`** operator return?

10.2.3. What is an automatic variable?

10.2.4. What is a dynamic variable?

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

```
// 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 = nullptr;
```

Figure 10.2: Resizing a dynamically allocated array

## 10.3 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, the value `nullptr` 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 10.2.

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 to 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_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 dynamic variables. This is done with the **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 *memory leak*.

After the old array is deallocated, da is set to point to the new array. Finally, da_new is set to the special pointer value nullptr, which is essentially a way of making da_new point to nothing.[2] This is done as a precaution so that daNew is not used later to accidentally modify or deallocate the array.

In the code of Figure 10.2, we copied the elements of da to da_new by writing our own loop. Could we have used the STL generic algorithm copy instead? This would require iterators and it turns out that pointers to array elements meet the requirements of random-access iterators: they support all the operations listed in Tables 6.1 and 6.2.

For example, pointers can be used to display the contents of array da as follows:

```
for (const int * p = da; p != da + n; ++p)
    cout << *p << ' ';
```

In this loop, p is initialized to point to the first element of the array. At every iteration, p is dereferenced to access the element it points to and then p is incremented to make it point to the next element. The loop terminates when p equals da + n, which is a pointer that points just past the last element of the array.

[2]The value nullptr is new to C++11. Before C++11, the value NULL was used instead. NULL can still be used but it is safer to use nullptr because nullptr is a pointer value while NULL is an integer and this can cause problems in some situations. The NULL value is defined in the standard libraries cstddef and cstdlib.

```
void init(int * a, int n)
// Initializes array a of size n to contain 0, 1,
// 2, ..., (n-1).
{
    for (int i = 0; i < n; ++i) a[i] = i;
}

void print_one_int(int x) { cout << x << ' '; }

void print(const int * a, int n)
// Prints the elements of array a of size n, on one
// line, separated by a single space.
{
    std::for_each(a, a + n, print_one_int);
    cout << '\n';
}
```

Figure 10.3: Functions that initialize and print an array

And now that we have array iterators, we can use the STL algorithms on arrays. For example, the contents of `da` can be copied to `da_new` as follows:

```
std::copy(da, da + n, da_new);
```

The passing of dynamically allocated arrays as arguments to functions raises a few particular issues. Figure 10.3 shows two functions `init` and `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.

The array argument could have also been declared as **`int`** `a[]`. This is equivalent to **`int`** `* a`. Note also that the argument `a` of `print` is declared as pointing to a constant integer. This prevents the function from modifying the contents of the array.

Note that the functions `init` and `print` can be used on ordinary arrays as well as on dynamically allocated arrays, as in

```
int a[5];
init(a, 5);
print(a, 5);
```

(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:

```
int * b = a;
```

This makes the pointer `b` point to array `a` (that is, 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,

```
std::fill(b, b+5, 0);
b[0] = 1;
print(b, 5);
```

fills the array with 0's, sets its first element to 1 and then prints the contents of the array.

Figure 10.4 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 by using the minimum of the old and the new sizes.

The functions `init`, `print` and `resize` can obviously be made generic, as shown in Figures 10.5 and 10.6. Note how requirements on the element type `T` are clearly documented. Note also how the template argument must

```
void resize(int * & da, int old_size, int new_size)
// Changes the size of dynamically allocated array da.
{
    int * da_new = new int[new_size];
    int min_size = std::min(old_size, new_size);
    std::copy(da, da + min_size, da_new);

    delete [] da;
    da = da_new;
    da_new = nullptr;
}
```

Figure 10.4: A function that resizes a dynamically allocated array

be specified when passing the function `print_one` to the generic algorithm `for_each`. Otherwise, the compiler would not know how to instantiate the function template when compiling the call to `for_each`.

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 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 apply to pointers or are related to the dynamic allocation of arrays. These operations are listed in Table 10.1. Also described in that table are several ways of

```
template <class T>
void init(T * a, int n)
// Initializes array a of size n to contain 0, 1,
// 2, ..., (n—1).
// 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_one(const T & x) { cout << x << ' '; }
// Assumption on T: values of type T can be printed to
// cout by by using the output operator (<<).

template <class T>
void print(const T * a, int n)
// Prints the elements of array a of size n, on one
// line, separated by a single space.
// Assumption on T: values of type T can be printed to
// cout by using the output operator (<<).
{
    std::for_each(a, a + n, print_one<T>);
    cout << '\n';
}
```

Figure 10.5: Generic array functions (part 1 of 2)

```
template <class T>
void resize(T * & da, int old_size, int new_size)
// Changes the size of dynamically allocated array da.
// Assumption on T: has a default constructor.
{
    T * da_new = new T[new_size];
    int min_size = std::min(old_size, new_size);
    std::copy(da, da + min_size, da_new);

    delete [] da;
    da = da_new;
    da_new = nullptr;
}
```

Figure 10.6: Generic array functions (part 2 of 2)

initializing dynamically allocated arrays.

## Study Questions

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

10.3.2. How can a dynamically allocated array be resized?

10.3.3. In the function `resize` shown in Figure 10.4, could the argument `da` be declared as `int * da`?

10.3.4. What is a memory leak?

## Exercises

10.3.5. Create a generic function

`T * p`
: Declares a pointer of type `T`.

`p = nullptr`
: Makes `p` point to nothing.

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

```
p = new T[n]
p = new T[n]()
p = new T[n]{}
p = new T[n]{elements}
```
: Makes `p` point to a new dynamically allocated array of type `T` and size `n`. The array elements are default-initialized in the first version, value-initialized in the second and third versions, and initialized to the given `elements` in the fourth version. In case the number of given `elements` is less than `n`, the remaining elements of the array are value-initialized.

`p[i]`
: Assuming that `p` points to an array element, returns a reference to the element `i` positions to the right of the element that `p` points to.

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

Table 10.1: Some operations related to pointers and the dynamic allocation of arrays

```
T * copy(const T * 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`.

10.3.6. Create a generic function

```
T * concatenate(const T * a, int n,
                const T * 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.

10.3.7. Consider the following arrays:

```
int a[5];
int b[5];
int * c = new int[5];
int * d = new int[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 11

# Implementation of Vectors

In this chapter, we will learn how to implement a basic class of vectors.

## 11.1 A Basic Class of Vectors

Figure 11.1 shows the declaration of a basic class of vectors. (Additional methods and operators will be added later in this section and in the following sections. You will be asked to add others in the exercises.)

The first thing to notice is that `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.

The class `Vector` has two data members. The first one is a pointer to a dynamically allocated array that contains the vector elements. It is common to call such an array a *buffer*. 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.

The method `size` simply returns the size of the vector. The implementation

```
template <class T>
class Vector
{
public:
    Vector() : buffer_(nullptr), size_(0) {}
    explicit Vector(int n);
    Vector(const std::initializer_list<T> & init_list);

    int size() const { return size_; }

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

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 nullptr if n == 0.
    T * get_new_buffer(int n) const;
};
```

Figure 11.1: The class declaration

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

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

but also modified, as in

```
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:

```
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 implementation of this constructor is shown in Figure 11.2. The constructor uses the private method `get_new_buffer`, whose implementation is also shown in Figure 11.2. The purpose of `get_new_buffer` is to ensure that the buffer is properly initialized to `nullptr` in case the given size is 0. (With some compilers, **`new`** may not return `nullptr` when the size is 0.)

Note that it is better for `get_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.

```
template <class T>
inline Vector<T>::Vector(int n)
{
    buffer_ = get_new_buffer(n);
    size_ = n;
}

template <class T>
inline T * Vector<T>::get_new_buffer(int n) const
{
    return (n == 0 ? nullptr : new T[n]());
}
```

Figure 11.2: The second constructor and the `get_new_buffer` method

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 11.3 shows the implementation of the constructor that takes an initializer list as argument. This is the constructor that is used in declarations such as

```
Vector<int> v1({10, 20, 30});
```

and

```
Vector<int> v1 = {10, 20, 30};
```

```
template <class T>
Vector<T>::Vector(
        const std::initializer_list<T> & init_list)
{
    buffer_ = get_new_buffer(init_list.size());
    std::copy(init_list.begin(), init_list.end(),
              buffer_);
    size_ = init_list.size();
}
```

Figure 11.3: The initializer-list constructor

or in implicit conversions such as `f({1, 2, 3})`, assuming that `f` takes a `Vector` as argument. Recall from Section 6.7 that initializer lists provide very limited functionality: they only support three methods: `size`, `begin` and `end`.

As explained in Section 2.9, the source code for a class is normally split into a header file and an implementation file. In the case of `Vector`, the header file `Vector.h` would contain the class declaration while the implementation file `Vector.cpp` would contain the implementation of all the methods that weren't implemented in the header file. The header file is then included in any source file that contains code that uses the class (a test driver, for example). The class and the code that uses the class can then be compiled separately because all the information the compiler needs about the class to be able to compile the client code is contained in the header file.

But there are exceptions to this basic setup. One concerns inline methods. When compiling code that uses inline methods, the compiler needs the body of those methods. As mentioned in Section 2.9, this implies that inline methods must be implemented in the header file. This is how our class `Time` was setup (see Figure 2.22).

Another exception concerns templates. Because `Vector` is a class template, its methods are also templates. And the compiler cannot fully compile the im-

plementation of a function template until it sees how the template is going to be instantiated. This requires knowing how the function is called. Therefore, the usual setup is to include the implementation of function templates in the header file, so that this code is available to the compiler when it is compiling code that uses the function template. In the case of a class template such as `Vector`, this means that the entire code of the class, the declaration of the class and the implementation of all the methods, is included in the header file. This is how the `Vector` source code available on the course web site is organized.

Source code and a test driver for the version of `Vector` presented in this section are available on the course web site under `Vector1.0`.

## Study Questions

11.1.1. Why is `get_new_buffer` declared private?

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

11.1.3. Why was the second constructor declared **`explicit`**?

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

## Exercises

11.1.5. Add the following methods, constructors and operators to `Vector`. They should behave just like their STL equivalents.

a) `empty()`.

b) `back()`. (Make sure to include both a constant and a non-constant version.)

c) `Vector(n, e)`.

d) Equality and inequality testing operators (==, !=).

e) `swap()`. (Make sure that no elements are copied.)

## 11.2 Iterators, Insert and Erase

The class of vectors we implemented in the previous section is fairly basic. It allows us to create vectors whose size is determined at run time but it doesn't allow us to change the size of those vectors, either by resizing the vector or by adding or removing elements from the vector. We will add this ability in this section. In particular, we will add the methods `insert` and `erase` to our class. (You will be asked to add `push_back`, `pop_back` and `resize` in the exercises.)

But first, we need to now add iterators to our vectors. A vector iterator should point to a vector element and support all the operations of random-access iterators. So we can simply use pointers to buffer elements to implement vector iterators. This works because these pointers point to the vector elements and also support all the operations of random-access iterators. Figure 11.4 shows the declaration of the types `iterator` and `const_iterator`, as well as the implementation of several `begin` and `end` methods. Note how a pointer that points past the last buffer element is used as the end iterator. The code of Figure 11.4 should be placed in the public section of the class.

Figure 11.9 shows an implementation of `erase`. We start by allocating a new buffer that's one smaller than the current size of the vector. The vector elements are then copied to the new buffer. Note how the element to be erased is skipped.

Figure 11.10 shows an implementation of `insert`. We start by allocating a new buffer that's one larger than the current size of the vector. The vector elements and the new element are then copied to the new buffer. Note how `new_itr` is set to point to the location of the new element in the new buffer.

```
typedef T * iterator;
typedef const T * const_iterator;

iterator begin() { return buffer_; }
const_iterator begin() const { return buffer_; }
const_iterator cbegin() const { return buffer_; }

iterator end() { return buffer_ + size_; }
const_iterator end() const { return buffer_ + size_; }
const_iterator cend() const { return buffer_ + size_; }
```

Figure 11.4: Iterators

```
template <class T>
inline typename Vector<T>::iterator Vector<T>::erase(
        Vector<T>::const_iterator const_itr)
{
    T * new_buffer = get_new_buffer(size_ - 1);
    iterator new_itr =
        std::copy(cbegin(), const_itr, new_buffer);
    std::copy(const_itr + 1, cend(), new_itr);
    delete [] buffer_;
    buffer_ = new_buffer;
    --size_;
    return new_itr;
}
```

Figure 11.5: The `erase` method

```
template <class T>
inline typename Vector<T>::iterator Vector<T>::insert(
        Vector<T>::const_iterator const_itr,
        const T & e)
{
    T * new_buffer = get_new_buffer(size_ + 1);
    iterator new_itr =
        std::copy(cbegin(), const_itr, new_buffer);
    *new_itr = e;
    std::copy(const_itr, cend(), new_itr + 1);
    delete [] buffer_;
    buffer_ = new_buffer;
    ++size_;
    return new_itr;
}
```

Figure 11.6: The `insert` method

These implementations of `insert` and `erase` are efficient in terms of memory space because they cause the vector to always have a buffer that's just large enough to store the current elements. But these implementations of `insert` and `erase` run in linear time because every time they are executed, every vector element must be copied to the new buffer. In contrast, the STL specifies that `insert` should run in amortized constant time and that `erase` should run in constant time. We will learn how to achieve these running times later in this chapter.

A version of `Vector` with the iterators and the implementations of `insert` and `erase` presented in this section is available on the course web site under `Vector1.1`.

### Exercises

11.2.1. Add the following methods to `Vector`. They should behave just like their STL equivalents (except in terms of their running times).

a) `push_back(e)`.

b) `pop_back()`.

c) `resize(n)`.

d) `resize(n, e)`.

## 11.3 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 will 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, a destructor has no return type. A `Vector` destructor can be declared and implemented as follows:

```
~Vector() { delete [] buffer_; }
```

Note that it is not necessary to ensure that the pointer is non-null before attempting to deallocate the buffer. That's because the **`delete`** operator will simply do nothing if given a null pointer.

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 begin 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 a **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.

1. When an object is created and initialized to be a copy of another one, through a declaration such as

   ```
   Vector<T> v1(v2);
   ```

   which can also be written as

```
Vector<T> v1 = v2;
```

2. When an object is passed by value to a function as in

```
void f(Vector<T> v)
```

3. When a function returns an object, as in

```
Vector<T> g()
{
  Vector<T> v;
  ...
  return v;
}
```

(But note that some compilers are able to avoid copying return values.)[1]

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*. 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 hold pointers that 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 11.7.

[1]This is done by essentially storing the local variable that is returned (`v` in this example) 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_ = get_new_buffer(v.size());
    std::copy(v.begin(), v.end(), buffer_);
    size_ = v.size();
}
```

Figure 11.7: A copy constructor

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 11.8. The 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 of Figure 11.8 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;
```

```
template <class T>
Vector<T> & Vector<T>::operator=(const Vector<T> & v)
{
    T * new_buffer = get_new_buffer(v.size());
    std::copy(v.begin(), v.end(), new_buffer);

    // deallocate old buffer
    delete [] buffer_;

    // give new buffer to receiver
    buffer_ = new_buffer;
    size_ = v.size();

    return *this;
}
```

Figure 11.8: An assignment operator

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

```
v1.operator=( v2.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 **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 dereferencing the pointer.

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 version of `Vector` available on the course web site under `Vector1.2` includes the destructor, copy constructor and assignment operator presented in this section.

## Study Questions

11.3.1. When is the destructor is called?

11.3.2. What are the three circumstances in which the copy constructor is called?

11.3.3. What kind of copy do the compiler-generated copy constructor and assignment operator perform?

11.3.4. Why doesn't our implementation of the assignment operator begin by deallocating the receiver's buffer?

11.3.5. Why does the assignment operator return its receiver?

11.3.6. In the body of a method, what does **`this`** refer to?

### Exercises

11.3.7. 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.

11.3.8. 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.)

11.3.9. Add the following methods to `Vector`. They should behave just like their STL equivalents.

a) `clear()`.

b) `assign(n, e)`.

## 11.4 Growing and Shrinking Vectors Efficiently

In the current version of our class `Vector`, the buffer is always just large enough to contain the elements of the vector. In other words, the size of the buffer is always equal to the size of the vector. As a consequence, every `insert` and

`erase` operation requires that we allocate a new buffer and copy all of the elements from the current buffer to the new one. All this copying implies that the running time of `insert` and `erase` is at least linear in the size of the vector. The same is true for `push_back` and `pop_back`.

Is there a way to avoid all this copying? And could this lead to a significant improvement in the running time of these operations? The answer is yes. In fact, the STL specifies that `pop_back` should run in constant time and that `push_back` should run in amortized constant time.

In contrast, in the current version of our class `Vector`, the amortized running time of both `pop_back` and `push_back` is linear. Here's why. suppose that we start with an empty vector and performs $n$ `push_back`'s. The first `push_back` allocates a buffer of size 1. No extra copying. But after that, whenever we allocate a buffer of size $i$, we copy $i-1$ elements from the current buffer to the new one. These copies add up to

$$1+2+\cdots+(n-1)=\frac{(n-1)n}{2}=\frac{n^2}{2}-\frac{n}{2}$$

Dividing by $n$, the number of elements inserted, we get an average of $n/2-1/2$ copies per element. That implies that the amortized running time of `push_back` is at least linear in $n$. A similar analysis gives the same result for `pop_back`.

Now, how can the STL running times be achieved? In the case of `pop_back`, it's not too difficult. The idea is to allow the buffer to be larger than the vector. To distinguish between the size of the buffer and the size of the vector, we will say that the size of the buffer is the **capacity** of the vector. And to keep the indexing operator simple, we will store the vector elements at the beginning of the buffer. This will ensure that the vector index of an element matches its buffer index.

Figure 11.9 shows an implementation of `erase` that uses this idea. (The `pop_back` operation can then be implemented by using `erase` or by adapting the implementation of `erase`.) The method starts by converting the argument

```
template <class T>
typename Vector<T>::iterator Vector<T>::erase(
        Vector<T>::const_iterator const_itr)
{
    iterator itr = iterator(const_itr);
    std::copy(itr + 1, end(), itr);
    --size_;
    return itr;
}
```

Figure 11.9: The `erase` method

from a constant iterator to a plain iterator. This is necessary so we can copy to the location that the iterator points to. The elements to the right of the element to be erased are then shifted one position to the left. Finally, the size of the vector is decreased by one, causing the size of the vector to be smaller than its capacity.

When implemented in this way, `pop_back` runs in constant time because no elements need to be copied. As for `erase`, in the worst case, that operation still runs in linear time because deleting from the beginning of the vector requires all the remaining elements to be shifted. However, in general, the running time of `erase` is still improved because elements to the left of the deletion point no longer need to be copied from the current buffer to a new one.

Getting the `push_back` operation to run in amortized constant time is a bit more difficult. The idea is to first notice that reallocations get increasingly expensive as the size of the vector increases. So we can try to ensure that as the vector grows, reallocations become less and less frequent. This can be done by adding an increasingly larger amount of capacity to the vector at each reallocation.

One strategy that works is to never increase the capacity by less than the current size. In other words, whenever the capacity needs to increase, we will

set the new capacity as follows:

$$\text{new\_capacity} = \max(\text{new\_size}, \text{current\_capacity} + \text{current\_size})$$

The impact of this strategy can be calculated. Suppose that we perform $n$ `push_back`'s into an initially empty vector. Whenever the buffer is reallocated, we have that the number of elements copied from the current buffer to the new one is equal to the current size. But

$$\text{new\_capacity} \geq \text{current\_capacity} + \text{current\_size}$$

which implies that

$$\text{new\_capacity} - \text{current\_capacity} \geq \text{current\_size}$$

Therefore,

$$\text{num\_copies} \leq \text{additional\_capacity}$$

Summing over every `push_back` that causes a reallocation, we get that

$$\text{total\_copies} \leq \text{final\_capacity}$$

Now, how large is the final capacity in comparison to the final size? The final capacity of the vector is the new capacity that was set at the last `push_back` that caused a reallocation. When that `push_back` occurred, the new capacity could have been set to the new size. If not, then

$$\text{new\_capacity} = \text{current\_capacity} + \text{current\_size}$$

But the fact that a reallocation happened implies that new size is greater than both the current size and the current capacity. Therefore,

$$\text{new\_capacity} \leq 2 \cdot \text{new\_size}$$

and this inequality holds in both cases.

Putting this together with the fact that the final capacity is the new capacity at the last reallocation, and that the new size at the last reallocation is no greater than the final size, we get that

$$\text{final\_capacity} \leq 2 \cdot \text{final\_size}$$

Therefore,

$$\text{total\_copies} \leq 2 \cdot \text{final\_size}$$

Dividing by the final size, which is the number of elements inserted, we get an average of at most 2 copies per element. By using this fact, it is possible to show that the amortized running time of `push_back` is constant. (We will learn how to do this in detail later in these notes.)

Note that the above analysis also holds when multiple elements are added in a single operation, as is the case with the `resize` operation. As for `insert`, in the worst case, that operation still runs in linear time because inserting at the beginning of the vector requires all the existing elements to be shifted. However, in general, the running time of `insert` can still be improved by the above strategy because it reduces the number of elements that are copied from the current buffer to a new one.

Figure 11.10 shows an implementation of `insert` that uses the above strategy. (Once again, the `push_back` operation can then be implemented by using `insert` or by adapting the implementation of `insert`.) Like `erase`, `insert` starts by converting the argument from a constant iterator to a plain iterator. Then, if the vector has enough capacity, elements are shifted to make room for the new one. Otherwise, a new buffer is allocated and the current elements are copied from the current buffer to the new one.

This implementation of `insert` assumes that the class has a new data member called `capacity_` that holds the current capacity of the vector. The implementation of several of the existing constructors, methods and operators need to be revised to set or update this data member.

The STL class `vector` provides three methods related to the capacity of vectors. These methods are described in Table 11.1. Note that if the final size

```
template <class T>
typename Vector<T>::iterator Vector<T>::insert(
        Vector<T>::const_iterator const_itr,
        const T & e)
{
    iterator itr = iterator(const_itr);

    if (size_ < capacity_) {
        std::copy_backward(itr, end(), end() + 1);
        *itr = e;
        ++size_;
        return itr;
    }
    else { // size_ == capacity_
        int new_capacity_ =
            capacity_ + std::max(1, size_);
        T * new_buffer = get_new_buffer(new_capacity_);
        iterator new_itr =
            std::copy(begin(), itr, new_buffer);
        *new_itr = e;
        std::copy(itr, end(), new_itr + 1);

        delete [] buffer_;
        buffer_ = new_buffer;
        ++size_;
        capacity_ = new_capacity_;
        return new_itr;
    }
}
```

Figure 11.10: The `insert` method

| | |
|---|---|
| `v.capacity()` | Asks vector `v` for its capacity. |
| `v.reserve(n)` | Asks vector `v` to increase its capacity so it is at least `n`. If `n` is less than the current capacity, nothing happens. |
| `v.shrink_to_fit()` | Asks vector `v` to reduce its capacity to the size of the vector. (Whether the request is fulfilled depends on the implementation.) |

Table 11.1: Vector operations related to the capacity of the vector

of a vector is known in advance, it is always more efficient to reserve enough capacity ahead of time than to let the vector manage its growth automatically.

The `capacity` method is easy to implement:

```
int capacity() const { return capacity_; }
```

The implementation of the other two methods is left as an exercise.

The version of `Vector` preented in this section is available on the course web site under `Vector2.0`.

## Exercises

11.4.1. Add the following methods to `Vector`. They should behave just like their STL equivalents, including with respect to their running times.

a) `push_back(e)`.

b) `pop_back()`.

c) `resize(n)`.

d) `resize(n, e)`.

e) `reserve(n)`.

f) `shrink_to_fit()`. (Make your implementation fulfill the request.)

# Chapter 12

# 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 and graphs.

## 12.1 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 12.1 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`

```
template <class T>
class ListNode
// T is the type of element stored in the list.
{
    friend class List<T>;

private:
    ListNode() :
        element(), next(nullptr), previous(nullptr) {}
    ListNode(const T & e,
             ListNode * next0,
             ListNode * previous0);

    T element;
    ListNode<T> * next;
    ListNode<T> * previous;
};

template <class T>
ListNode<T>::ListNode(const T & e,
                      ListNode * next0,
                      ListNode * previous0) :
        element(e), next(next0), previous(previous0) {}
```

Figure 12.1: A class of nodes

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 constructors and data members private and grant friendship only to `List`.

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 12.2. 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 dummy head node, and the previous pointer of the dummy head node point to the last node. This will make the list **circular** and ensure that every node,

```
template <class T>
class List
// T is the type of element stored in the List.
{
private:
    ListNode<T> * p_head_node;
};
```

Figure 12.2: The class `List` with a pointer to its head 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

12.1.1. Why can't we store a list's nodes inside the list object?

12.1.2. What is a dummy head node?

12.1.3. What is the advantage of having a dummy head node and making the list circular?

## 12.2 Some Basic Methods

We are now ready to start adding operations to our class `List`. These operations will form a small but representative subset of the STL `list` operations. You will be asked to implement additional operations in the exercises.

The implementation of `List` will be done gradually. In this section, we start with a default constructor, `push_back`, `pop_back` and `back`. Figure 12.3 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 12.4. The method *traverses* the linked list by using a pointer `p_node` 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 *dereference-and-select operator*. Recall that this is equivalent to `(*p_head_node).next`.

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 12.5 shows 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`.

```
class List
// T is the type of element stored in the list.
{
public:
    List();

    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 12.3: A first version of `List`

```
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 12.4: The internal test driver `test_print`

```
template <class T>
inline List<T>::List()
{
    p_head_node = new ListNode<T>;
    p_head_node->next =
        p_head_node->previous = p_head_node;
}
```

Figure 12.5: The default constructor

```
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>(new_element, p_head_node,
                        p_last_node);

    // finish linking new node to list
    p_last_node->next = p_new_node;
    p_head_node->previous = p_new_node;
}
```

Figure 12.6: The `push_back` method

Figure 12.3 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.

Implementations of `push_back` and `pop_back` are shown in Figures 12.6 and 12.7. 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`.

```
template <class T>
inline void List<T>::pop_back()
{
    // set pointers to last node and node that will
    // become last
    ListNode<T> * p_last_node = p_head_node->previous;
    ListNode<T> * p_new_last_node =
        p_last_node->previous;

    // modify the list to skip the last node
    p_new_last_node->next = p_head_node;
    p_head_node->previous = p_new_last_node;

    // deallocate the last node
    p_last_node->next =
        p_last_node->previous = nullptr;
    delete p_last_node;
}
```

Figure 12.7: The `pop_back` method

## Exercises

12.2.1. Add the following methods, constructors and operators to `List`. They should behave just like their STL equivalents.

a) `size()`. (To implement this method efficiently, add a new data member to the class to keep track of the current size of the list.)

b) `empty()`.

c) `front()`. (Make sure to include both a constant and a non-constant version.)

d) `push_front(e)`.

e) `pop_front()`.

f) `List(n)`.

g) `List(n, e)`.

h) `List(init_list)`.

i) `clear()`.

j) Equality and inequality testing operators (==, !=). (As usual, it is better to implement these operators as standalone functions so that implicit conversions can occur on both sides of the operators. This requires that at least one of the operators be a friend of both `ListNode` and `List`. This can be done by putting an advance declaration of the operator before the declaration of those two classes and then adding to the classes a friendship declaration similar to this one:

```
friend bool operator==<T>(
    const List<T> & ls1,
    const List<T> & ls2);
```

Note how the template parameter must be explicitly listed. Otherwise, the compiler would have no way of knowing that the friendship declaration refers to a function template.)

k) `swap()`. (Make sure that no elements are copied.)

12.2.2. If, in the previous exercise, you used `push_back` or `push_front` to implement the constructors `List(n)` and `List(n, e)`, revise those implementations to avoid using those methods. If done carefully, these implementations can be more efficient. *Hint*: If you add new elements at the end of the list, there is no need to link the last node with the head node after every insertion.

## 12.3 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 not where the next element is located.

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 may very well not be 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, as shown in Figure 12.8.

```
template <class T>
class ListIterator
// T is the type of element stored in the List.
{
public:
    ListIterator() : p_current_node(nullptr) {}

    ...

private:
    explicit 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 12.8: `ListIterator`

We're including two constructors in our iterator class. One is a default constructor that initializes the iterator to contain a null pointer. We know that it's good practice not to leave variables with unspecified values. But the STL also requires that default iterators evaluate to equal when compared by using the == operator. Having these iterators all contain a null pointer will make it easy to achieve this.

The second constructor is a private helper constructor that will allow lists to easily create iterators with specific pointers. (We will see examples of this later in this section.) Note that this constructor is declared **`explicit`** so that all conversions from pointers to iterators must be done explicitly. This is slightly less convenient than allowing implicit conversions but it may be a little safer and lead to code that's a bit easier to understand.

Figure 12.9 shows the implementation of the dereferencing operator (`*`), the derefencing-and-select operator (`->`) and the equality and inequality testing operators (`==`, `!=`).

The overloading of the dereferencing-and-select operator requires some explanation. This operator is used when the list elements are objects or structures. For example, if `itr` points to an element in a list of `Time`'s, then `itr->hours()` will return the hours of the time. Now, when the compiler sees the operator used as

```
itr->member
```

it will interpret it as

```
(itr.operator->())->member
```

This implies that we need to make our overloaded -> operator return a *pointer* to the list element. The implementation shown in Figure 12.8 achieves this by using the **address-of** operator (`&`). This unary operator returns a pointer to its operand.

Figure 12.10 shows the implementation of the prefix ++ operator. The operator needs to return the receiver so that code such as

```
T & operator*() const
{
    return p_current_node->element;
}

T * operator->() const
{
    return &(p_current_node->element);
}

bool operator==(const ListIterator & rhs) const
{
    return (p_current_node == rhs.p_current_node);
}

bool operator!=(const ListIterator & rhs) const
{
    return (p_current_node != rhs.p_current_node);
}
```

Figure 12.9: The operators *, ->, == and =! of class `ListIterator`

```
template <class T>
inline ListIterator<T> & ListIterator<T>::operator++()
// prefix version (++itr)
{
    p_current_node = p_current_node->next;
    return *this;
}
```

Figure 12.10: The prefix ++ operator of class `ListIterator`

```
template <class T>
inline ListIterator<T> ListIterator<T>::operator++(int)
// postfix version (itr++)
{
    ListIterator<T> original_itr = *this;
    p_current_node = p_current_node->next;
    return original_itr;
}
```

Figure 12.11: The postfix ++ operator of class `ListIterator`

```
itr1 = ++itr2;
```

can work properly. Recall that in a method, the variable **`this`** always points to the receiver.

Figure 12.11 shows the implementation of the postfix ++ operator. Note how a dummy integer argument is used to distinguish between the prefix and postfix versions of this operator. In addition, note how the postfix version saves a copy of the iterator before incrementing it.

Most iterator operations need to access the three `ListNode` members `element`, `next` and `previous`. A convenient way of making this possible without making these data members public is to have `ListNode` grant friendship to `ListIterator`.

To complete the addition of iterators to our class of linked lists, we need to define the `iterator` type and implement the methods `begin` and `end`. Recall that the users of STL lists expect that list iterators will be of type

```
list<T>::iterator
```

Therefore, we include in our class `List` the following declaration:

```
typedef ListIterator<T> iterator;
```

```
iterator begin()
{
    return iterator(p_head_node->next);
}
iterator end() { return iterator(p_head_node); }
```

Figure 12.12: The `begin` and `end` methods

This will cause `List<T>::iterator` to mean `ListIterator<T>`.

The `begin` and `end` methods can be implemented as shown in Figure 12.12 In each of these methods, an iterator is constructed from a pointer to a node by using the private constructor of class `ListIterator`. We can allow these methods to access this constructor by making `List` a friend of `ListIterator`.

Constant iterators can be added to `List` by duplicating the `ListIterator` code and making some small modifications to it. Figure 12.13 shows the needed changes. They mostly consist in declaring that pointers and references point or refer to constant elements. But note that the class also contains an additional constructor that allows a non-constant iterator to be converted into a constant iterator. (But we don't allow the reverse conversion.) The implementation of this constructor requires that `ConstListIterator` be made a friend of `ListIterator`.

We also need to add to `List` the declaration of the type `const_iterator`, the constant versions of `begin` and `end`, as well as the methods `cbegin` and `cend`. These are shown in Figure 12.14.

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. This function is shown in Figure 12.15. Note that we can use a range-for loop on a constant list such as `ls` only because we added constant iterators to our class of linked lists.

Figures 12.16 and 12.17 show the implementation the `insert` and `erase`

```
template <class T>
class ConstListIterator
// T is the type of element stored in the List.
{
   friend class List<T>;

public:
    ConstListIterator(const ListIterator<T> & itr) :
        p_current_node(itr.p_current_node) {}

    const T & operator*() const
    {
        return p_current_node->element;
    }
    const T * operator->() const
    {
        return &(p_current_node->element);
    }

    ...

private:
    explicit ConstListIterator(const ListNode<T> * p) :
        p_current_node(p) {}

    const ListNode<T> * p_current_node;
        // points to the node that contains the
        // element that the iterator currently
        // points to
};
```

Figure 12.13: The declaration of `ConstListIterator`

```
typedef ConstListIterator<T> const_iterator;

const_iterator begin() const
{
    return const_iterator(p_head_node->next);
}
const_iterator end() const
{
    return const_iterator(p_head_node);
}

const_iterator cbegin() const
{
    return const_iterator(p_head_node->next);
}
const_iterator cend() const
{
    return const_iterator(p_head_node);
}
```

Figure 12.14: The `const_iterator` type and the constant versions of `begin` and `end`

```
template <typename T>
void print(const List<T> & ls)
{
    for (const T & e : ls) cout << e << ' ';
    cout << '\n';
}
```

Figure 12.15: A function that prints a `List`

methods. These implementations use the fact that `List` can access the private members of the node and iterator classes. Once again, the code implementing these methods is easier to follow by drawing pictures. Note how the pointer of the constant iterator received as argument must be cast to a non-constant pointer to allow us to modify the node.

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

12.3.1. What type of value should an overloaded operator –> return?

12.3.2. Why does the iterator operator ++ return its receiver?

## Exercises

12.3.3. Add the following methods and operators to `List`. They should behave just like their STL equivalents.

a) Equality and inequality testing operators (==, `!=`). (If you did this exercise in the previous section, you did it using pointers. Redo it now using iterators. But don't use the STL algorithm `equal` because our `List` iterators do not meet all the requirements for interacting properly with STL algorithms.)

b) `remove(e)`.

c) `erase(start, stop)`.

d) `reverse()`. (Make sure your implementation works for lists of odd and even length. No iterators should be invalidated.)

```
template <class T>
inline typename List<T>::iterator List<T>::insert(
        const_iterator const_itr,
        const T & new_element)
{
    // set pointer to the node that should follow the
    // new one
    ListNode<T> * p_next_node =
        (ListNode<T> *)(const_itr.p_current_node);

    // set pointer to the node that should precede the
    // new one
    ListNode<T> * p_previous_node =
        p_next_node->previous;

    // create new node and set its contents
    ListNode<T> * p_new_node =
        new ListNode<T>(new_element,
                        p_next_node, 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 12.16: The `insert` method

```
template <class T>
inline typename List<T>::iterator List<T>::erase(
        const_iterator const_itr)
{
    // set pointer to the node to be deleted
    ListNode<T> * p_target_node =
        (ListNode<T> *)(const_itr.p_current_node);

    // set pointers to the nodes that precede and
    // follow the target node
    ListNode<T> * p_previous_node =
        p_target_node->previous;
    ListNode<T> * p_next_node = p_target_node->next;

    // 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 =
        p_target_node->previous = nullptr;
    delete p_target_node;

    return iterator(p_next_node);
}
```

Figure 12.17: The `erase` method

```
template <class T>
List<T>::~List()
{
    // erase all elements
    while (begin() != end()) pop_back();

    // deallocate dummy head node
    delete p_head_node;
}
```

Figure 12.18: The destructor

12.3.4. If, in the previous exercise, you implemented `erase(start, stop)` by calling `erase(itr)` repeatedly, revise the implementation to avoid doing that. If done carefully, this implementation can be more efficient.

## 12.4 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 destructor is shown in Figure 12.18. 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 copy constructor is shown in Figure 12.19. After initializing the list to be empty, through delegation to the default construc-

```
template <class T>
List<T>::List(const List<T> & ls) : List<T>()
{
    for (const T & e : ls) push_back(e);
}
```

Figure 12.19: A 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 12.20: An assignment operator

tor, the copy constructor simply goes through the argument list and adds a copy of each of its elements to the back of the receiver.

An assignment operator is shown in Figure 12.20. 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 exchange the nodes of the copy with those of the receiver by using the STL 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 entire source code for our class of linked lists is available on the course web site under `List1.2`.

## Exercises

12.4.1. Revise the destructor so it uses the `size` or `empty` methods you implemented for an exercise earlier in this chapter.

12.4.2. Implement the destructor without using other methods such as `pop_back`. If done carefully, a more direct implementation can be more efficient.

12.4.3. Implement the copy constructor without using other methods such as `push_back`. If done carefully, a more direct implementation can be more efficient.

# Chapter 13

# 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.

## 13.1 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*.

In these notes, we already considered efficiency when choosing a data structure to store data in the text editor and phone book programs. Later in these notes, efficiency will be the main consideration when we learn algorithms for searching and sorting data.

In this chapter, we will 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

13.1.1. What are some of the properties quality software should have?

13.1.2. What does it mean for software to be efficient?

13.1.3. What two computational resources are most often considered?

[1] Algorithms that use randomness are usually studied in a course such as CS447 *Computer Algorithms*.

## 13.2 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.

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 13.1 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

| $n$ | 10 | $10^3$ | $10^6$ |
|---|---|---|---|
| $\log_2 n$ $\mu s$ | 3 $\mu s$ | 10 $\mu s$ | 20 $\mu s$ |
| $n$ $\mu s$ | 10 $\mu s$ | 1 ms | 1 s |
| $n^2$ $\mu s$ | 100 $\mu s$ | 1 s | 12 days |

Table 13.1: Running times of three algorithms

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 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

13.2.1. When comparing the efficiency of algorithms, why is it usually important to compare running times over a wide range of input sizes?

```
for i = 0 to n−1
    print a[i]
```

Figure 13.1: Printing the contents of an array

13.2.2. What are three significant weaknesses of comparing algorithms by measuring exact running times?

## 13.3 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 13.1 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]

[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

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 13.1. 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.

---

operations typically provided by programing 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

13.3.1. As described in this section, what does analysis seek to determine?

13.3.2. What are three advantages and two disadvantages of analysis over the measurement of exact running times through testing?

## 13.4 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 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 analyzing an algorithm, 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 13.1** *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 13.2** *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 "$f(n)$ is $\Theta(g(n))$" 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, and to summarize, 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 its 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

[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 $\lim_{n\to\infty}(h(n)/g(n)) = 0$.

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

13.4.1. What is the asymptotic running time of an algorithm?

13.4.2. How exactly does the asymptotic running time of an algorithm relate to its exact running time?

13.4.3. What does it mean for two running times to be asymptotically equivalent?

13.4.4. What is the main advantage of simplifying the running time of an algorithm?

13.4.5. What is asymptotic analysis?

## Exercises

13.4.6. Below is 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) $3n^2 - n + 10 = \Theta(n^2)$.

13.4.7. 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.)

## 13.5 Some Common Running Times

Table 13.2 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 13.2 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 13.1). Table 13.3 provides some numbers that compare linear, quadratric 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.

### Exercises

13.5.1. To each of Tables 13.1 and 13.3, add rows for the running times $n \log_2 n$ and $n^3$.

13.5.2. How large does $n$ have to be before $2^n$ is larger than each of the following functions: $n$, $n^2$, $n^3$ and $n^6$?

| RUNNING TIME | COMMON NAME | TYPICAL EXAMPLE |
|---|---|---|
| $\Theta(1)$ | constant | a single basic operation |
| $\Theta(\log n)$ | logarithmic | fast searching algorithms |
| $\Theta(n)$ | linear | simple searching algorithms |
| $\Theta(n \log n)$ | $n \log n$ | fast sorting algorithms |
| $\Theta(n^2)$ | quadratic | simple sorting algorithms |
| $\Theta(n^k)$ | polynomial | most algorithms that are fast enough to be useful in practice |
| $\Theta(c^n)$, where $c > 1$ | exponential | exhaustive searches of very large sets |
| $\Theta(n!)$ | factorial | same as above |

Table 13.2: Some common running times

| $n$ | 10 | 20 | 40 | 60 | 80 |
|---|---|---|---|---|---|
| $n$ $\mu s$ | 10 $\mu s$ | 20 $\mu s$ | 40 $\mu s$ | 60 $\mu s$ | 80 $\mu s$ |
| $n^2$ $\mu s$ | 0.1 ms | 0.4 ms | 1.6 ms | 3.6 ms | 6.4 ms |
| $2^n$ $\mu s$ | 1 ms | 1 s | 13 days | $37 \times 10^3$ years | $38 \times 10^9$ years |

Table 13.3: More execution times

## 13.6 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 13.3** 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

$$T(n) = \Theta(n) + \Theta(1) + \Theta(n)$$

This simply means that $T(n)$ is the sum of three functions and these functions are $\Theta(n)$, $\Theta(1)$ and $\Theta(n)$, respectively.

It should be clear that one of the $\Theta(n)$ functions will dominate and therefore that $T(n)$ is $\Theta(n)$. □

The conclusion at the end of this example can be justified informally by referring to the definition of the $\Theta$ notation, as follows. The fact that the three functions are $\Theta(n)$, $\Theta(1)$ and $\Theta(n)$ tells us that there are positive constants $b_1$, $b_2$ and $b_3$ such that

$$T(n) \leq b_1 n + b_2 + b_3 n$$

Therefore,

$$T(n) \leq b_1 n + b_2 n + b_3 n = (b_1 + b_2 + b_3)n$$

On the other hand, the fact that the three functions are $\Theta(n)$, $\Theta(1)$ and $\Theta(n)$ also tells us that there are positive constants $a_1$, $a_2$ and $a_3$ such that

$$T(n) \geq a_1 n + a_2 + a_3 n$$

Therefore,

$$T(n) \geq a_1 n + a_3 n = (a_1 + a_3)n$$

So we have that

$$an \leq T(n) \leq bn$$

where $a = a_1 + a_3$ and $b = b_1 + b_2 + b_3$. This means that $T(n) = \Theta(n)$.

The above argument is informal because it ignores the $n_0$ constant in the definition of the $\Theta$ notation. In other words, the argument ignores the fact that these inequalities don't hold for every value of $n$ but only when $n$ is large enough.

But it is possible to carry out these arguments more formally. In fact, it is possible to prove that the $\Theta$ notation is both *additive* and *transitive*.

**Property 13.4 (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 the previous example, since $T(n) = \Theta(n) + \Theta(1) + \Theta(n)$, additivity can be used to conclude that $T(n) = \Theta(2n + 1)$.

**Property 13.5 (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 $g(n)$ and $g(n)$ is about the same $h(n)$, then $f(n)$ should be about the same $h(n)$.

In the previous example, since $T(n) = \Theta(2n+1)$ and $2n+1 = \Theta(n)$, additivity can be used to conclude that $T(n) = \Theta(n)$. To summarize,

$$\begin{aligned} T(n) &= \Theta(n) + \Theta(1) + \Theta(n) \\ &= \Theta(n + 1 + n) \text{ (by additivity)} \\ &= \Theta(2n + 1) \\ &= \Theta(n) \text{ (since } 2n + 1 = \Theta(n) \text{ and by transitivity)} \end{aligned}$$

**Example 13.6** 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

$$\begin{aligned} T(n) &= \Theta(n) + \Theta(n^2) + \Theta(1) \\ &= \Theta(n + n^2 + 1) \\ &= \Theta(n^2) \end{aligned}$$

□

**Example 13.7** Besides consecutive steps, loops are another very common form of algorithm. Figure 13.2 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 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

$$T(n) = c_{\text{init}} + c_{\text{test}} + n(c_{\text{body}} + c_{\text{update}} + c_{\text{test}})$$

Clearly, $T(n) = \Theta(n)$. □

```
for i = 0 to n−1
    print a[i]
```

Figure 13.2: Printing the contents of an array

```
for i = 0 to n−1
    for j = 0 to n−1
        print a[i,j]
```

Figure 13.3: Printing the contents of a two-dimensional array

**Example 13.8** Figure 13.3 shows an algorithm that consists of two nested loops. It prints the contents of a two-dimensional array. The usual strategy for analyzing 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, since the body of the outer loop is the inner loop, we have that $t_{\text{body}}(n) = T_{\text{inner}}(n) = \Theta(n)$. It seems clear that this implies that $nt_{\text{body}}(n) = \Theta(n^2)$ and therefore that $T_{\text{outer}}(n) = \Theta(n^2)$. □

```
for i = 0 to n−1
    for j = 0 to i−1
        print a[i,j]
```

Figure 13.4: Printing the lower left triangle of a two-dimensional array

Once again, the conclusion at the end of the previous example can be justified informally by referring to the definition of the $\Theta$ notation. The fact that $t_{\text{body}}(n) = \Theta(n)$ tells us that there are constants $a$ and $b$ such that

$$an \leq t_{\text{body}}(n) \leq bn$$

Therefore,

$$an^2 \leq nt_{\text{body}}(n) \leq bn^2$$

This means that $nt_{\text{body}}(n) = \Theta(n^2)$.

This argument can be made more formal and used to prove another property of the $\Theta$ notation, the fact that it is *multiplicative*.

**Property 13.9 (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 example, since $n = \Theta(n)$ and $t_{\text{body}}(n) = \Theta(n)$, multiplicativity can be used to conclude that $nt_{\text{body}}(n) = \Theta(n^2)$. Then, since $T_{\text{outer}}(n) = \Theta(nt_{\text{body}}(n))$, we get that $T_{\text{outer}}(n) = \Theta(n^2)$, by transitivity.

**Example 13.10** Let's analyze a slightly more complicated loop. Figure 13.4 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 inner loop repeats $i$ times instead of $n$ times. This implies means that the running time of the inner loop varies with $i$. In fact, the running time of the inner loop is $\Theta(i)$. This has the important consequence

that we cannot simply multiply the running time of the inner loop by the number of times the outer loop repeats. Instead, we need to *add* the running time of all the executions of the inner loop.

Here's how we can do this. Let $T_{\text{inner}}(i)$ be the running time of the inner loop. Since the operations that manage the loop run in constant time, we have that

$$T_{\text{outer}}(n) = \Theta\left(\sum_{i=0}^{n-1} T_{\text{inner}}(i)\right)$$

Now, since $T_{\text{inner}}(i) = \Theta(i)$, there is a constant $b$ such that the running time of the inner loop is bounded above by $bi$. (Once again, we are being informal by ignoring the fact that this upper bound holds only when $i$ is large enough.) Therefore,

$$\sum_{i=0}^{n-1} T_{\text{inner}}(i) \leq \sum_{i=0}^{n-1} bi = b\sum_{i=0}^{n-1} i$$

We can then use the well-known formula $1 + 2 + \cdots + k = k(k+1)/2$:

$$b\sum_{i=0}^{n-1} i = b\frac{(n-1)n}{2} = \frac{b}{2}n^2 - \frac{b}{2}n$$

Therefore,

$$\sum_{i=0}^{n-1} T_{\text{inner}}(i) \leq \frac{b}{2}n^2 - \frac{b}{2}n$$

A similar argument shows that this sum is also bounded below by a quadratic function. Therefore, the running time of the outer loop is $\Theta(n^2)$. □

**Example 13.11** 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 13.5 shows a hybrid sorting algorithm that takes advantage of that fact.

```
if (n < 10)
    sort using simple sorting algorithm
else
    sort using fast sorting algorithm
```

Figure 13.5: A hybrid sorting algorithm

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 13.12 (Reflexivity)** *Every function is asymptotically equivalent to itself. That is, every function $f(n)$ is $\Theta(f(n))$.*

**Property 13.13 (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 on discrete mathematics or in a course on mathematics for computer science.[4]

## Exercises

13.6.1. 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.

[4] At Clarkson, this is MA211 *Foundations*.

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 '*'
```

13.6.2. Consider the algorithm shown in Figure 13.6. Let $T_A(n)$, $T_B(n)$ and $T_C(n)$ denote the running time of algorithms $A$, $B$ and $C$, respectively. What is the running time of this algorithm, as a function of $n$, under each of the following sets of assumptions? Don't forget to simplify and use the $\Theta$ notation. Justify your answers.

a) $T_A(n) = \Theta(n)$, $T_B(n) = \Theta(n^2)$ and $T_C(n) = \Theta(\log n)$.

```
A
if (n < 100)
    B
else
    for j = 1 to n
        C
```

Figure 13.6: Algorithm for Exercise 13.6.2

```
for i = 0 to n−1
    if (a[i] == x) return i
return −1
```

Figure 13.7: A sequential search of an array

b) $T_A(n) = \Theta(n^2)$, $T_B(n) = \Theta(n^2)$ and $T_C(n) = \Theta(\log n)$.

c) $T_A(n) = \Theta(n^2)$, $T_B(n) = \Theta(n^3)$ and $T_C(n) = \Theta(\log n)$.

13.6.3. Consider the class of vectors we implemented earlier in these notes. Analyze the running time of the various operations. For each function, 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.

## 13.7 Worst-Case and Average-Case Analysis

Consider the sequential search algorithm shown in Figure 13.7. 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

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 $\Theta(k)$, where $k$ is the index of the first occurrence of $x$. As usual, there is a constant $n$ such that this running time is at most $bk$. An upper bound on the average running time can then be obtained by taking the average over all possible positions $k$:

$$\frac{1}{n}\sum_{k=0}^{n-1} bk = \frac{b}{n}\sum_{k=0}^{n-1} k = \frac{b}{n}\frac{(n-1)n}{2} = b\frac{n-1}{2}$$

A similar argument also gives a linear lower bound. Therefore, the average running time of a successful search is $\Theta(n)$.

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 required 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.

This apparent contradiction can be explained in part 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.[5] 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 algorithm 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 "empirical

[5] We will do the worst-case analysis of quicksort later in these notes. At Clarkson, the average-case analysis is usually done in the course CS344 *Algorithms and Data Structures*.

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

13.7.1. What are the best-case, worst-case and average-case running times of an algorithm?

13.7.2. What is an advantage and a disadvantage of the worst-case running time compared to the average-case running time?

### Exercises

13.7.3. 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.

## 13.8 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.

```
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 13.8: The binary search algorithm

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 sequence. Because the sequence is sorted, this comparison tells us if the element 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 13.8. Figure 13.9 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 13.10 shows a generic implementation of the binary search algorithm

```
e = 25
s = [12  16  25  37  38  42  60  73]   middle = 38
    [12  16  25  37]                           25
             [25  37]                          37
             [25]
Found!
```

Figure 13.9: A run 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 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 keep things simple, we'll assume that $n$ is a power of 2.

Consider the pseudocode shown in Figure 13.8. 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 $T(n) = \Theta(\log n)$.

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.

```
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.
{
    while (stop - start >= 2) {
        int middle = (start + stop) / 2;
        if (e < a[middle])
            stop = middle;
        else
            start = middle;
    }
    if (stop - start == 1) {
        if (e == a[start])
            return start;
        else
            return -1;
    }
    else {  // stop - start <= 0
        return -1;
    }
}
```

Figure 13.10: An implementation of binary search for arrays

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.)

## Exercises

13.8.1. 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 13.9.

13.8.2. 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.) Show that the running time of the binary search algorithm is linear in this case.

13.8.3. 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 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.

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

```
void print1(int n, char c)
{
    for (int i = 0; i < n; ++i) {
        cout << c;
    }
    cout << endl;
}
```

Figure 14.1: A simple iterative algorithm

`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:

```
cout << c;
print(n−1, c);
```

The function call `print(n−1, c)` is *recursive* because it occurs in `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 *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.

```
void print2(int n, char c)
{
    if (n > 0) {
        cout << c;
        print2(n-1, c);
    }
    else {
        cout << endl;
    }
}
```

Figure 14.2: A recursive algorithm

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:

> `print(n, c)` correctly prints a line containing $n$ copies of `c` when you assume that `print(n-1, c)` correctly prints a line containing $n-1$ copies of `c`.

In our case, this statement is true. But why does it guarantee that `print(n, c)` works for every possible value $n$? The key is to consider what the property says for $n$, $n-1$, all the way down to 1:

> `print(n, c)` works if `print(n-1, c)` works
> `print(n-1, c)` works if `print(n-2, c)` works
> `print(n-2, c)` works if `print(n-3, c)` works

...
`print(2, c)` works if `print(1, c)` works
`print(1, c)` works if `print(0, c)` 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:

`print(1, c)` works because `print(0, c)` works
`print(2, c)` works because `print(1, c)` works
...
`print(n-2, c)` works because `print(n-3, c)` works
`print(n-1, c)` works because `print(n-2, c)` works
`print(n, c)` works because `print(n-1, c)` works

Therefore, `print(n, c)` works.

This type of argument can be expressed more formally by using the *Principle of Mathematical Induction*.[1]

It is important to realize that each recursive call executes independently from the others. In particular, each recursive call has its own arguments and its own set of local variables. For example, the execution of `print(2, '*')` 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 13.8). But the algorithm can also

[1] At Clarkson, this proof technique is covered in the course MA211 *Foundations*.

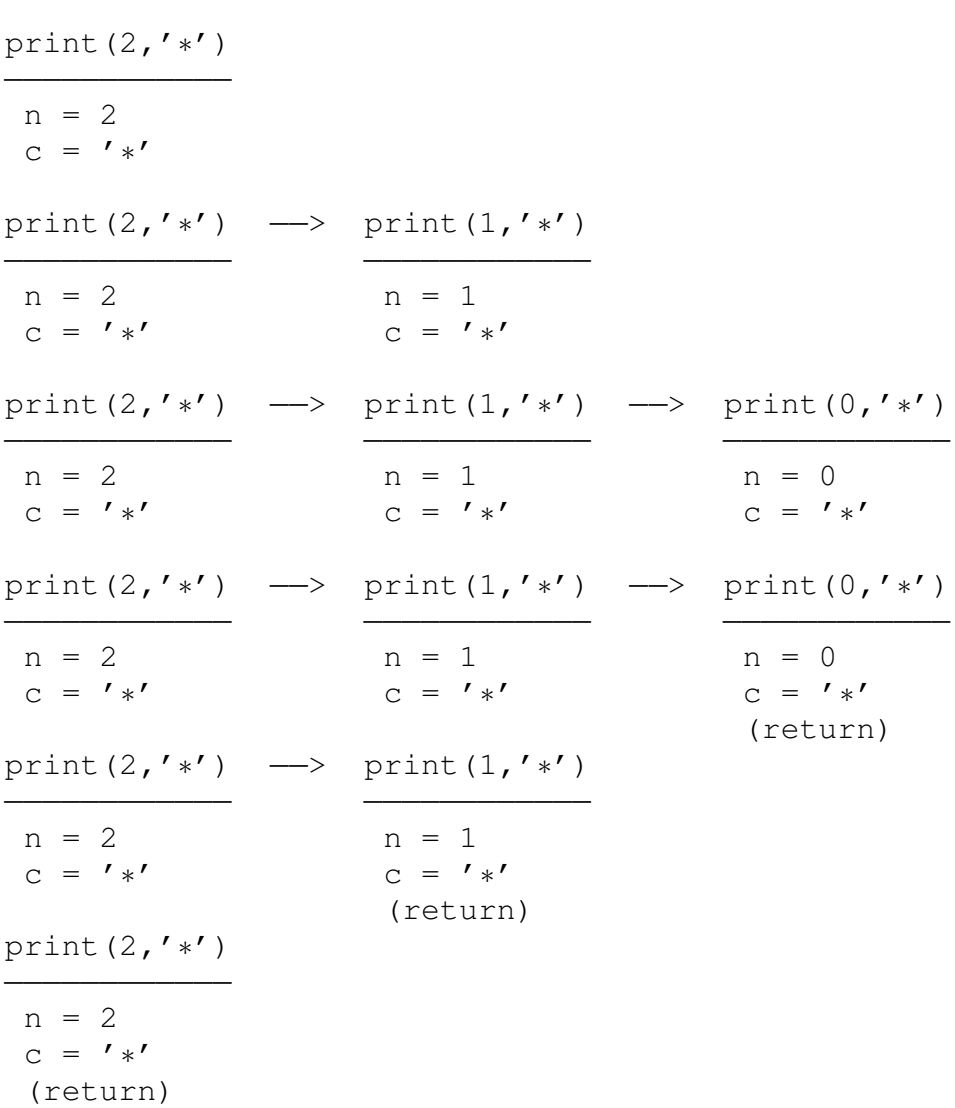


Figure 14.3: A sample run of the `print` algorithm

```
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

```
template <class T>
void display(const T a[], int start, int stop)
// Displays 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) {
        cout << a[start] << ' ';
        display(a, start+1, stop);
    }
    // if start >= stop, do nothing
}
```

Figure 14.5: Implementation of the recursive display algorithm

```
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

```
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

be described recursively, as shown in Figure 14.10. The idea is that after 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.

## 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,

```
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.
{
    if (stop - start >= 2) {
        int middle = (start + stop) / 2;
        if (e < a[middle])
            return binary_search(a, start, middle, e);
        else
            return binary_search(a, middle, stop, e);
    }
    if (stop - start == 1) {
        if (e == a[start])
            return start;
        else
            return -1;
    }
    else {  // stop - start <= 0
        return -1;
    }
}
```

Figure 14.11: An implementation of the recursive binary search for arrays

a `start` index, a `stop` index and an element. The function returns the number of times the element occurs in the range `[start,stop)`.

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 $n, n-1, \ldots, 3, 2, 1$. The function should do nothing if $n < 1$.

14.1.7. Repeat the previous question but this time print the numbers in increasing order.

14.1.8. Repeat again, this time printing $n, n-1, \ldots, 3, 2, 1, 2, 3, \ldots, n-1, n$. 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 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 of 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.[2]

### 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 `binary_search` are tail recursive while `sum` and `display_reverse` 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]At Clarkson, these languages are typically studied in the course CS341 *Programming Languages*.

```
void print(int n, char c)
{
    while (n > 0) {
        cout << c;
        --n;
    }
    cout << endl;
}
```

Figure 14.12: A version of `print` with the tail recursion removed

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

```
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

```
[60  12  37  42  25  38  16]
[16  12  37  42  25  38] 60
[12  16  25  37  38  42] 60
[12  16  25  37  38  42  60]
```

Figure 15.2: A run of the selection sort algorithm (top level of recursion)

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 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

```
[60  12  37  42  25  38  16]
[16  12  37  42  25  38] 60
[16  12  37  38  25] 42  60
[16  12  37  25] 38  42  60
[16  12  25] 37  38  42  60
[16  12] 25  37  38  42  60
[12] 16  25  37  38  42  60
[12] 16  25  37  38  42  60
[12  16] 25  37  38  42  60
[12  16  25] 37  38  42  60
[12  16  25  37] 38  42  60
[12  16  25  37  38] 42  60
[12  16  25  37  38  42] 60
[12  16  25  37  38  42  60]
```

Figure 15.3: A run of the selection sort algorithm (entire recursion)

```
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 15.4: 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) {
        std::swap(*std::max_element(a + start,
                                    a + stop),
                  a[stop - 1]);
        --stop;
    }
}
```

Figure 15.5: An implementation of selection sort for arrays

is shown in Figure 15.5. Note the use of the STL 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,

$$\begin{aligned} T(n) &= T(n-1)+\Theta(n) \quad \text{(when } n \geq 2\text{)} \\ T(1) &= \Theta(1) \end{aligned}$$

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 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, replace the asymptotics with actual functions:

$$\begin{aligned} T(n) &\leq T(n-1)+bn \quad (\text{when } n \geq 2) \\ T(1) &= c \end{aligned}$$

Then, note that the recurrence relation implies the following set of equations:

$$\begin{aligned} T(n) &\leq T(n-1)+bn \\ T(n-1) &\leq T(n-2)+b(n-1) \\ &\vdots \\ T(2) &\leq T(1)+b2 \\ T(1) &= c \end{aligned}$$

Second, add of all these equations to get

$$\begin{aligned} T(n) &\leq c+b(2+\cdots+n) \\ &= c+b\left(\frac{n(n+1)}{2}-1\right) \end{aligned}$$

This means that $T(n)$ is bounded above by a quadratic function. A similar argument shows that $T(n)$ is also bounded below by a quadratic function. This implies that $T(n)=\Theta(n^2)$.

```
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

### 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[stop−1]` into `[start,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

```
[60  12  37  42  25  38  16]
[60  12  37  42  25  38] 16
[12  25  37  38  42  60] 16
[12  16  25  37  38  42  60]
```

Figure 15.7: A run of the insertion sort algorithm (top level of recursion)

shows the final contents of the array, after the insertion of the last element into the sorted subarray.

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

```
[60 12 37 42 25 38 16]
[60 12 37 42 25 38] 16
[60 12 37 42 25] 38 16
[60 12 37 42] 25 38 16
[60 12 37] 42 25 38 16
[60 12] 37 42 25 38 16
[60] 12 37 42 25 38 16
[60] 12 37 42 25 38 16
[12 60] 37 42 25 38 16
[12 37 60] 42 25 38 16
[12 37 42 60] 25 38 16
[12 25 37 42 60] 38 16
[12 25 37 38 42 60] 16
[12 16 25 37 38 42 60]
```

Figure 15.8: A run of the insertion sort algorithm (entire recursion)

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,

$$\begin{aligned} T(n) &= T(n-1)+\Theta(n) \quad \text{(when } n \geq 2\text{)} \\ T(1) &= \Theta(1) \end{aligned}$$

where the $\Theta(n)$ term is essentially the running time of the insertion. 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  37  25  60  16  42  38
```

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. 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.

```
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

```
[60   12   37   42   25   38   16]
[60   12   37   42] [25   38   16]
[12   37   42   60] [16   25   38]
[12   16   25   37   38   42   60]
```

Figure 15.10: A run of the mergesort algorithm (top level of recursion)

```
[60   12   37   42   25   38   16]
[60   12   37   42] [25   38   16]
[60   12] [37   42] [25   38] [16]
[60] [12] [37] [42] [25] [38] [16]
[12   60] [37   42] [25   38] [16]
[12   37   42   60] [16   25   38]
[12   16   25   37   38   42   60]
```

Figure 15.11: A run of the mergesort algorithm (entire recursion)

```
 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

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`. Figures 15.14 and 15.15 show a possible implementation of the merging algorithm.

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 a 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

```
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);

        std::inplace_merge(a + start, a + middle,
                           a + stop);
    }
}
```

Figure 15.13: An implementation of mergesort for arrays

```
template <class T>
void merge(T a[], int start, int middle, int stop)
// Merges two sorted, consecutive subarrays.  Runs in
// linear time.
//
// PRECONDITION: The indices are valid and occur in
// the following order: start, middle, stop.  The
// elements in each of the ranges [start,middle) and
// [middle,stop) are sorted in increasing order.
// There is enough memory available for allocating
// stop - start values of type T.
//
// POSTCONDITION: The elements in the range [start,
// stop) are sorted in increasing order according to
// the < operator.
//
// ASSUMPTION ON TEMPLATE ARGUMENT: Values of type T
// can be compared using the < operator.
{
    int i1 = start; // index of next element to be
                    // considered in first subarray
    int i2 = middle; // same for second subarray

    T * result = new T[stop - start];
        // temporary array to store merged elements
    int j = 0; // index of next available position in
               // result

    ...
}
```

Figure 15.14: An implementation of the merging algorithm (part 1 of 2)

```
template <class T>
void merge(T a[], int start, int middle, int stop)
{
    ...

    while (i1 < middle && i2 < stop) {
        if (a[i1] < a[i2]) {
            result[j] = a[i1];
            ++i1;
        } else {
            result[j] = a[i2];
            ++i2;
        }
        ++j;
    }
    // At this point, either i1 == middle or i2 ==
    // stop, which means that at least one of the two
    // subarrays has been completely copied to result.

    // Copy the rest of the first subarray to result.
    std::copy(a + i1, a + middle, result + j);

    // Copy the rest of the second subarray to result.
    std::copy(a + i2, a + stop, result + j);

    // Copy result to original array
    std::copy(result, result + (stop - start),
              a + start);

    delete [] result;
}
```

Figure 15.15: An implementation of the merging algorithm (part 2 of 2)

linear time. Therefore,

$$\begin{aligned} T(2^k) &= 2T(2^{k-1}) + \Theta(2^k) \quad \text{(when } k \geq 1\text{)} \\ T(1) &= \Theta(1) \end{aligned}$$

To solve this recurrence relation, we start by replacing the asymptotics by actual functions:

$$\begin{aligned} T(2^k) &\leq 2T(2^{k-1}) + b2^k \quad \text{(when } k \geq 1\text{)} \\ T(1) &= c \end{aligned}$$

We then consider the following series of equations:

$$\begin{aligned} T(2^k) &\leq 2T(2^{k-1}) + b2^k \\ T(2^{k-1}) &\leq 2T(2^{k-2}) + b2^{k-1} \\ &\vdots \\ T(2) &\leq 2T(1) + b2 \\ T(1) &= c \end{aligned}$$

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 multiply each equation by an appropriate constant:

$$\begin{aligned} T(2^k) &\leq 2T(2^{k-1}) + b2^k \\ 2T(2^{k-1}) &\leq 2^2T(2^{k-2}) + b2^k \\ 2^2T(2^{k-1}) &\leq 2^3T(2^{k-2}) + b2^k \\ &\vdots \\ 2^{k-1}T(2) &\leq 2^kT(1) + b2^k \\ 2^kT(1) &= c2^k \end{aligned}$$

Now, if we add all of these, we get

$$T(2^k) \leq kb2^k + c2^k$$

Since $n = 2^k$, we have that $k = \log n$, so that

$$T(n) \leq bn \log n + cn$$

A similar argument shows that

$$T(n) \geq an \log n + cn$$

Therefore, since the $n \log n$ terms dominate, we get that $T(n) = \Theta(n \log n)$.

Note that what each equation

$$2^i T(2^{k-i}) \leq 2^i T(2^{k-i-1}) + b2^k$$

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.

## 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:

```
  First array       Second array
[16 22 25 37 60]    [19 38 42 46]
```

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 possible about the elements to be sorted. One possibility is to only require 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: every comparison-based algorithm has a running time that's at least $\Theta(n \log n)$. Even on average.[1]

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, recursively 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

[1] At Clarkson, this result is usually proven in CS344 *Algorithms and Data Structures*.

```
input: an array

if (array contains more than one element) {
    choose a pivot element
    partition the array around the pivot
    sort each subarray
}
```

Figure 15.16: The quicksort algorithm

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 15.16. 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.

Figure 15.17 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. Element 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 15.18 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

```
[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]
```

Figure 15.17: A run of the quicksort algorithm (top level of recursion)

```
[60   12   37   42   25   38   16]
[12   37   25   38   16]  42  [60]
[12]  16  [37   25   38]  42  [60]
[12]  16  [25]  37  [38]  42  [60]
[12]  16  [25   37   38]  42  [60]
[12   16   25   37   38]  42  [60]
[12   16   25   37   38   42   60]
```

Figure 15.18: A run of the quicksort algorithm (entire recursion)

certain level were executed simultaneously. The elements that were chosen as pivots are 42, 16 and 37. The second half of Figure 15.18 shows, for every level of the recursion, the final contents of the array, after the recursive sorting of the subarrays.

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 algorithm is more complicated and no more efficient than the recursive version. Therefore, quicksort is another good example of recursion.

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. We

won't describe the algorithm in these notes, but the median can be found in linear time.[2] This leads to the following recurrence relation for the running time of quicksort, assuming that the size of the array is a power of 2:

$$\begin{aligned} T(2^k) &= 2T(2^{k-1}) + \Theta(2^k) \quad \text{(when } k \geq 1\text{)} \\ T(1) &= \Theta(1) \end{aligned}$$

The $\Theta(2^k)$ term is essentially the computation of the median and the partitioning of the array. This is the same recurrence relation we had for mergesort. Therefore, quicksort runs in time $\Theta(n \log n)$ when the median element is used as pivot. However, the linear-time algorithm that computes the median is not that simple, with the consequence that the hidden constant factors are somewhat large. So it's unlikely that this version of quicksort would run faster than mergesort.

A much faster option is to simply pick the first element of the array as pivot. But if the array is already sorted, the smallest element would be used as pivot, leading to the most uneven partition possible. What happens next depends on the details of the partitioning step. For example, suppose that elements are not reordered unnecessarily during the partitioning. Then, the array will be partitioned into an empty array on one side and an ordered array on the other. This implies that the smallest element will always be used as pivot and that every partition will be as uneven as possible. In this case, the running time of quicksort will be given by the following recurrence relation:

$$\begin{aligned} T(n) &= T(n-1) + \Theta(n) \quad \text{(when } n \geq 2\text{)} \\ T(1) &= \Theta(1) \end{aligned}$$

where the $\Theta(n)$ term is essentially the time it takes to partition the array. This is the same recurrence relation that gave the running time of selection sort. Recall that its solution is $\Theta(n^2)$.

In practice, it is not that uncommon for at least large portions of the input array to already be sorted. Therefore, using the first element as pivot would lead

[2] At Clarkson, this algorithm is usually covered in CS344 *Algorithms and Data Structures*.

quicksort to often run slowly. Similar problems occur when the last or middle elements are used as pivots.

Another idea is to use as pivot the median of the first, middle and last elements of the array. If the array is sorted, then this would cause the median to be used as pivot, leading to the best possible partitions and an optimal running time of $\Theta(n \log n)$. But it's not hard to construct arrays that would be unevenly partitioned under this choice of pivot. (See one of the exercises.)

The safest option is probably to use a random element as pivot. This gives a bad partition only if the pivot is among the smallest or largest elements of the array. The probability that this should happen very often should be small. In fact, it is possible to show that using a random element as pivot leads to an average running time of $\Theta(n \log n)$. The average is taken over all the possible pivot choices and is the same for every input array.[3]

An algorithm that uses randomness is said to be a **randomized** algorithm. Even though the randomized version of quicksort has an average running time of $\Theta(n \log n)$, it is still possible that the algorithm would always chose the minimum element as pivot while running on some array. This is unlikely but possible. And in that case, the running time of quicksort would once again degenerate to the $\Theta(n^2)$ running time of selection sort.

Because this running time occurs when the partitions are as uneven as possible, it is tempting to conclude that the worst-case running time of quicksort is $\Theta(n^2)$. But the truth is we haven't ruled out the possibility that quicksort may run even slower for some other type of partitions. Fortunately, it turns out that our intuition is correct: it is possible to show that the worst-case running time of quicksort is $\Theta(n^2)$.[4]

It is important to point out that even though the worst-case running time of quicksort is $\Theta(n^2)$, which is as bad as the running time of selection sort and insertion sort, in practice, quicksort tends to run faster than even mergesort.

[3] At Clarkson, this result is usually proven in CS344 *Algorithms and Data Structures*.

[4] At Clarkson, this is another result that is usually proven in CS344 *Algorithms and Data Structures*.

The fact that the average running time of quicksort is $\Theta(n \log n)$, which is the same as mergesort, explains why this is possible.

## 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 15.17. Then show the entire recursion, as in Figure 15.18. 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.

# Bibliography


[Bro87] Fred P. Brooks. No silver bullet: Essence and accidents of software engineering. *Computer*, 20(4):10–19, 1987.

[CPP] cppreference.com. Web. Last accessed September 2015. `http://cppreference.com`.

[Goo] Google C++ Style Guide. Web. Last accessed September 2015. `http://google-styleguide.googlecode.com/svn/trunk/cppguide.xml`.

[LB03] Craig Larman and Victor R. Basili. Iterative and incremental development: A brief history. *Computer*, 36(6):47–56, 2003.

[Sim] Charles Simonyi. Hungarian notation. *MSDN*. Microsoft, 1999. Web. Last accessed August 2015. `http://msdn.microsoft.com/en-us/library/Aa260976`.

[Str] Bjarne Stroustrup. The C++ programming language. Web. Last accessed October 2013. `http://www.stroustrup.com/C++.html`.

[TA] Turing Award. `http://www.acm.org/awards/taward.html`.

# Index