Introduction to Problem Solving and Programming through C++

Abhiram Ranade

Contents

Preface

Learning to program is like learning a natural language in many ways. A few stages can be identified in the process of language learning: (a) being able understand spoken or written language, (b) being able to speak or write the language, (c) write/speak formally, (d) ability to understand the literature of that language (e) write/speak creatively. In some ways these stages are present even in learning programming languages.

The first stage of programming language acquisition is understanding the syntax and semantics of the language. Mastery of this basically enables the learner to simulate the execution of an arbitrary program using pen and paper. It is not expected that the learner infers the *intent* in any way; the learner merely knows what effect the execution of each statement has, and can thus step through execution, keeping track of the program state by writing it down on paper if needed. This is by no means a trivial skill, especially given the complexity of languages such as C++. But even if we consider a simple subset of the language (or consider simpler languages), clearly understanding ideas such as control flow, storage allocation, scope rules, is a noteworthy achievement for any learner.

The next stage, analogous to speaking a natural language, is one in which the learner starts writing programs. Of course, the two stages are not temporally disjoint. Indeed, when a learner speaks a language she is not only communicating her thoughts, but also putting up her composition for criticism. Through such interaction, her comprehension skills are affirmed, in addition to expression skills. Likewise, someone learning C++ might write a simple program having just a single `for` statement to see if her understanding of the statement is correct. Such experiments are extremely valuable for language acquisition, and they come for free if you have a computer. A learner must be encouraged to try such experiments from day one of the learning process.

There is of course more to expressing yourself in a language. For programming languages, the analogous situation is as follows. Suppose you have learned the Newton-Raphson method of finding roots in a course in Mathematics. Can you write a program to find the roots? What is required here is the ability to translate from the informal description of an algorithm learned in the mathematics classroom into the constructs provided by the programming language. The informal language using which algorithms are learned in mathematics may not match the programming language. This can be because of a variety of reasons. For example, there is no equivalent of a programming language variable in traditional mathematics. To make matters worse, there exists a notion of a variable in mathematics which is different and hence can cause confusion. Another difficulty is presented by looping constructs: the standard constructs have a loop exit test at the beginning or at the end, whereas the natural description of most processes require a loop exit somewhere in the middle. This can either be handled using a `break` construct, or by hoisting some code out of the loop (often the more favoured style). These are some of the skills to be learnt.

The third stage is analogous to formal communication in natural languages. This can be metaphorically considered to be equivalent to the ability to reason formally about programs. On the one hand this involves the ability to write down clear specifications (as opposed to having an intuitive idea) of the problem being solved, and reasoning with these specifications. On the other hand, this leads to various paradigms of program organization, with particular emphasis on object oriented programming. The former opens up the entire area of formal verification of programs. Given that formal verification is not practised by professional programmers, it is debatable how much this should be stressed in an introductory course. We feel, however, that even a beginner should understand that in principle, programs must be proved correct, and that there exist techniques for doing this. Notions such as preconditions, postconditions and invariants must be understood. It is felt that these ideas will subconsciously guide the student into writing better programs, even if they are found too laborious to implement rigorously at the present time. As to programming paradigms, our approach is somewhat pragmatic. At every point in the book we have tended to choose a programming style which appears most appropriate for the job at hand, given the tools the student has. From time to time we have given alternate ways of writing a program. However, there is a progression towards object oriented programming. It has happened as a byproduct of the requirements, rather than because of the goal of following a stylistic dictat.

The fourth stage is analogous to reading good literature. What is the point of learning a new language if not to read something exciting written in it? For programming this means acquiring some familiarity with some classical algorithms. We have made a conscious effort in this book to acquaint the reader with interesting computational problems in a variety of areas, from mathematics and physical sciences to operations research. Of course, this barely scratches the plethora of elegant algorithms that could be exciting for the reader. Our hope is merely that it will create a taste and excitement in the reader to seek out computations and algorithms. We will have served our purpose, for example, if this book causes readers to want to learn more about cosmological simulation, or optimizing airport layouts, or data structures or error analysis in numerical algorithms.

The final stage, the ability to design new algorithms, is perhaps analogous to creative writing in natural language. Algorithm design is a vast subject, and there are huge tomes written on it. But we feel that an introduction to programming and problem solving must contain a description of the most primitive and yet perhaps the most powerful problem solving idea: recursion. We have attempted to give a somewhat detailed introduction to recursion using several examples. Recursion is important not only as an algorithm design tool, but also as a mechanism for expressing programs: several programs are more elegant when expressed recursively. We have also considered memoization as a natural optimization for recursion, and so we could have said to have touched upon the technique of dynamic programming as well. The second theme in our presentation concerns the use of exhaustive search. Exhaustive search can be expressed cleanly using recursion, and hence it is a good testing ground to exercise recursion. More importantly, we feel that it reassures the student of the power of computer programs by expanding his vision of what can be solved using computers.

Most of the book, enough to learn basics of C++, is meant to be accessible to students who have passed standard X. Some sections are addressed more to science and engineering students. However, I would like to mention that when I taught a course based on this material several students came up and said that the programming exercises related to advanced mathematics topics actually helped them understand the mathematics topics better. So even if you feel mathematics is not your strong point, you may still want to read the mathematically involved sections of the book with an optimistic attitude.

I feel that a course in programming should excite the problem solver in you, and you should feel a sense that everything is solvable, not just grand problems but even mundane problems (more useful and often as difficult!). If this book can encourage you to view day to day problems as computation, and give you the confidence that you can write programs to tackle them, this book will have served its purpose.

Chapter 1

Introduction

A computer is one of the most remarkable machines invented by man. Most other machines have a very narrow purpose. A watch shows time, a camera takes pictures, a truck carries goods from one point to another, an electron microscope shows magnified views of very small objects. Some of these machines are much larger than a computer, and many much more expensive, but a computer is much, much more complex and interesting in the kind of uses it can be put to. Indeed, many of these machines, from a watch to an electron microscope typically might contain a computer inside them, performing some of the most vital functions of each machine. The goal of this book is to explain how a computer can possibly be used for so many purposes, and many more.

Viewed one way, a computer is simply an electrical circuit; a giant, complex electrical circuit, but a circuit nevertheless. In principle, it is possible to make computers without the use of electricity – indeed there have been designs of computers based on using mechanical gears, or fluidics devices.[1] But all that is mostly of historical importance. For practical purposes, today, it is fine to regard a computer as an electrical circuit. Parts of this circuit are capable of receiving data from the external world, remembering it so that it can be reproduced later, processing it, and sending the results back to the external world. By data we could mean different things. For example, it could mean some numbers you type from the keyboard of a computer. Or it could mean electrical signals a computer can receive from a sensor which senses temperature, pressure, light intensity and so on. The word process might mean something as simple as calculating the average of the sequence of numbers you type from the keyboard. It could also mean something much more complex: e.g. determining whether the signals received from a light sensor indicate that there is some movement in the vicinity of the sensor. Finally, by "send data to the external world" we might mean something as simple as printing the calculated average on the screen of your computer so that you can read it. Or we could mean activating a beeper connected to your computer if the movement detected is deemed suspicious. Exactly which parts of the circuit are active at what time is decided by a *program* fed to the computer.

It is the program which distinguishes a computer from most other machines; by installing different programs the same computer can be made to behave in dramatically different ways. How to develop these programs is the subject of this book. In this chapter, we will begin

[1] Also it is appropriate to think of our own brain as a computer made out of biological material, i.e. neurons or neural cells.

by seeing an example of a program. It turns out that we can understand, or even develop (typically called *write*) programs without knowing a lot about the specific circuits that the computer contains. This is very similar to how one might learn to drive a car; clearly one can learn to drive without knowing how exactly an automobile engine works. So you will be able to not only understand the program that we show you, but yourself write some programs.

There are many languages using which programs can be written. The language we will use in this book is the C++ programming language, invented in the early 1980s by Bjarne Stroustrup. For the initial part of the book, we will not use the bare C++ language, but instead augment it with a package called `simplecpp` (which stands for simple C++) which we have developed. How to install this package is explained in Appendix A. We developed this package so that C++ appears more friendly and more fun to people who are starting to learn C++. To use the driving metaphor again, it could be said that C++ is like a complex racing car. When you are learning to drive, it is better to start with a simpler vehicle, in which there aren't too many confusing controls. Also, standard C++ does not by default contain the ability to draw pictures. The package `simplecpp` does contain this feature. We thus expect that by using the `simplecpp` package it will be easier and more fun to learn the language. But in a few chapters, you will outgrow `simplecpp` and be able to use standard C++ (like "the pros"), unless of course you are using the graphics features.

1.1 A simple program

Our first example program is given below.

```
#include <simplecpp>
main_program{
  turtleSim();

  forward(100);
  left(90);
  forward(100);
  left(90);
  forward(100);
  left(90);
  forward(100);

  wait(5);
  closeTurtleSim();
}
```

If you execute this program on your computer, it will first open a window. Then a small triangle which we call a *turtle*[2] will appear in the window. Then the turtle will move and draw lines as it moves. The turtle will draw a square and then stop. After that, the window will vanish, and the program will end. Shortly we will describe how to execute the program.

[2] Our turtle is meant to mimic the turtle in the Logo programming language.

First we will tell you why the program does all that it does, and this will help you modify the program to make it do something else if you wish.

The first line `#include <simplecpp>` declares that the program makes use of some facilities called `simplecpp` in addition to what is provided by the C++ programming language.

The next line, `main_program{`, says that what follows is the main program.[3] The main program itself is contained in the braces { } following the text `main_program`.

The line following that, `turtleSim()` causes a window with a triangle at its center to be opened on the screen. The triangle represents our turtle, and the screen the ground on which it can move. Initially, the turtle points in the East direction. The turtle is equipped with a pen, which can either be raised or lowered to touch the ground. If the pen is lowered, then it draws on the ground as the turtle moves. Initially, the pen of the turtle is lowered, and it is ready to draw.

The next line `forward(100)` causes the turtle to move forward by the amount given in the parentheses, (). The amount is to be given in *pixels*. As you might perhaps know, your screen is really an array of small dots, each of which can be of any colour. Typical screens have an array of about 1000 × 1000 dots. Each dot is called a pixel. So the command `forward(100)` causes the turtle to go forward in the current direction it is pointing by about a tenth of the screen size. Since the pen was down, this causes a line to be drawn.

The command `left(90)` causes the turtle to turn left by 90 degrees. Other numbers could also be specified instead of 90. After this, the next command is `forward(100)`, which causes the turtle to move forward by 100 pixels. Since the turtle is facing north this time, the line is drawn northward. This completes the second side of the square. The next `left(90)` command causes the turtle to turn again. The following `forward(100)` draws the third side. Then the turtle turns once more because of the third `left(90)` command, and the fourth `forward(100)` finally draws the fourth side and completes the square.

After this the line `wait(5)` causes the program to do nothing for 5 seconds. This is the time you have to admire the work of the turtle!

Finally the last line `closeTurtleSim()` causes the window to be removed from the screen. After executing this line, the program halts.

Perhaps you are puzzled by the `()` following the commands `turtleSim` and `closeTurtleSim`. The explanation is simple. A command in C++ will typically require additional information to do its work, e.g. for the `forward` command, you need to specify a number denoting how far to move. It just so happens that `turtleSim` and `closeTurtleSim` do not require any additional information. Hence we need to simply write `()`. Later you will see that there can be commands which will need more than one pieces of information, in this case we simply put the pieces inside `()` separated by commas.

1.1.1 Executing the program

To execute this program, we must first have it in a file on your computer. It is customary to use the suffix `.cpp` for files containing C++ programs. So let us suppose you have typed the program into a file called `square.cpp` – you can also get the file from the CD or the book webpage.

[3] Yes, there can be non-main programs too, as you will see later.

Next, we must *compile* the file, i.e. translate it into a form which the computer understands more directly and can *execute*. The translation is done by the command `s++` which got installed when you installed the package `simplecpp`. The command `s++` merely invokes the GNU C++ compiler, which must be present on your computer (See Section A). In a UNIX shell you can compile a file by typing `s++` followed by the name of the file. In this case you would type `s++ square.cpp`. As a result of this another file is produced, which contains the program in a form that is ready to execute. On UNIX, this file is typically called `a.out`. This file can be executed by typing its name to the shell

```
% a.out
```

You may be required to type `./a.out` because of some quirks of UNIX. Or you may be able to execute by double clicking its icon. When the program is thus executed, you should see a window come up, with the turtle which then draws the square.

1.2 Remarks

A C++ program is similar in many ways to a paragraph written in English. A paragraph consists of sentences separated by full stops; a C++ program contains commands which must be separated by semi-colons. Note that while most human beings will tolerate writing in which a full-stop is missed, a computer is very fastidious, each command must be followed by a semi-colon. Note however, that the computer is more forgiving about the use of spaces and line breaks. It is acceptable to put in spaces and linebreaks almost anywhere so long as words or numbers are not split or joined. Thus it is perfectly legal (though not recommended!) to write

```
turtleSim();forward(100)    ;
left  (90
);
```

if you wish. This flexibility is meant to enable you to write such that the program is easy to understand. Indeed, we have put empty lines in the program so as to help ourselves while reading it. Thus the commands which actually draw the square are separated from those that open and close the windows. Another important idea is to *indent*, i.e. put leading spaces before lines that are part of `main_program`. This is again done to make it visually apparent what is a part of the main program and what is not. As you might observe, indentation is also used in normal writing in English.

1.2.1 Execution order

Another important similarity concerns the order of the sentences and commands. A paragraph is expected to be read from left to right, top to bottom. So is a program. By default a computer executes the commands left to right, top to bottom. But just as you have directives in magazines or newspaper such as “Please continue from page 13, column 4”, the order in which the commands of a program are executed can be changed. We see an example next.

1.3 Repeating a block of commands

At this point you should be able to write a program to draw any regular polygon, say a decagon. You need to know how much to turn at each step. The amount by which you turn equals the exterior angle of the polygon. But we know from Euclidean Geometry that the exterior angles of a polygon add up to 360 degrees. A decagon has 10 exterior angles, and hence after drawing each side you must turn by $360/10 = 36$ degree. So to draw a decagon of side length 100, we repeat the `forward(100)` and `right(36)` commands 10 times. This works, but you may get bored writing down the same command several times. Indeed, you dont need to do that. Here is what you would write instead.

```
#include <simplecpp>
main_program{
  turtleSim();
  repeat(10){
        forward(100);
        left(36);
  }
  closeTurtleSim();
}
```

This program, when executed, will draw a decagon. The new statement in this is the `repeat`. Its general form is

```
repeat(count){
  statements
}
```

In this, `count` could be any number. The `statements` could be any sequence of statements which would be executed as many times as the expression `count`, in the given order. The `statements` are said to constitute the *body* of the `repeat` statement. Each execution of the body is said to be an *iteration*. Only after the body of the loop is executed as many times as the value of `count`, do we execute the statement following the `repeat` statement.

So in this case the sequence `forward(100); left(36);` is executed 10 times, drawing all 10 edges of the decagon. Only after that we execute `closeTurtleSim()`.

1.3.1 Drawing any regular polygon

Our next program when executed, asks the user to type in how many sides the polygon should have, and then draws the required polygon.

```
#include <simplecpp>
main_program{
  int nsides;
  cout << "Type in the number of sides: ";
  cin >> nsides;
  turtleSim();

  repeat(nsides){
        forward(50);
        left(360.0/nsides);
  }
  wait(5);
  closeTurtleSim();
}
```

This program has a number of new ideas. The first statement in the main program is `int nsides;` which does several things. The first word `int` is short for “integer”, and it asks that a region be reserved in memory in which integer values will be stored during execution. Second, it also gives a name to this region: `nsides`. Finally it declares that from now on, whenever the programmer uses the name `nsides` it should be considered to refer to this region. It is customary to say that `nsides` is a variable, whose value is stored in the associated region of memory. This statement is said to *define* the variable `nsides`. As many variables as you want can be defined, either by giving separate definition statements, or by writing out the names with commas in between. For example, `int nsides, length;` would define two variables, the first called `nsides`, the second `length`. We will learn more about names and variables in the next chapter.

The next new statement is relatively simple. `cout` is a name that refers to the computer screen. It is customary to pronounce the `c` in `cout` (and `cin` in the next statement) as “see”. The sequence of characters `<<` denotes the operation of writing something on the screen. What gets written is to be specified after the `<<`. So the statement in our program will display the message

Type in the number of sides:

on the screen. Of course, you may put in a different message in your program, and that will get displayed.

In the statement after that, `cin >> nsides;`, the name `cin` refers to the keyboard. It asks the computer to wait until the user types in something from the keyboard, and whatever is typed is placed into the (region associated with) the variable `nsides`. The user must type in an integer value followed by typing the return key. The value typed in gets placed in `nsides`.

After the `cin >> nsides;` statement is executed, the computer executes the `repeat` statement. Executing a repeat statement is nothing but executing its body as many times as specified. In this case, the computer is asked to execute the body `nsides` times. So if the user had typed in 15 in response to the message asking for the number of sides to be typed, then the variable `nsides` would have got the value 15, and the loop body would be executed 15 times. The loop body consists of the two statements `forward(100)` and `left(360.0/nsides)`. Notice that instead of directly giving the number of degrees to turn, we have given an expression. This is allowed! The computer will evaluate the expression, and use that value. Thus in this case the computer will divide 360.0 by the value of the variable `nsides`, and the result is the turning angle. Thus, if `nsides` is 15, the turning angle will be 24. So it should be clear that in this case a 15 sided polygon would be drawn.

1.3.2 Repeat within a repeat

What do you think the program below does?

```
#include <simplecpp>
main_program{
  int nsides;

  turtleSim();

  repeat(10){
    cout << "Type in the number of sides: ";
    cin >> nsides;
    repeat(nsides){
        forward(50);
        left(360.0/nsides);
    }
  }
  closeTurtleSim();
}
```

The key new idea in this program is the appearance of a `repeat` statement inside another `repeat` statement. How does a computer execute this? Its rule is simple: to execute a repeat statement, it just executes the body as many times as specified. In each iteration of the outer repeat statement there will one execution of the inner repeat statement. But one execution of the inner repeat could have several iterations. Thus, in this case a single iteration of the outer repeat will cause the user to be asked for the number of sides, after the user types in the number, the required number of edges will be drawn by the inner repeat statement. After that, the next iteration of the outer repeat would begin, for a total of 10 iterations. Thus a total of 10 polygons would be drawn, one on top of another.

1.4 Some useful turtle commands

The following commands can also be used.

`penUp()`: This causes the pen to be raised. So after executing this command, the turtle will move but no line will be drawn until the pen is lowered. There is nothing inside the () because no number is needed to be specified, as was the case with `forward`, e.g. `forward(10)`.

`penDown()`: This causes the pen to be lowered. So after executing this command, a line will be drawn whenever the turtle moves, until the pen is raised again.

Thus if you write `repeat(10){forward(10); penUp(); forward(5); penDown();}` a dashed line will be drawn.

1.5 Numerical functions

The commands you have seen so far for controlling the turtle will enable you to draw several interesting figures. However you will notice that it is cumbersome to draw some simple figures. For example, if you wish to draw an isoceles right angled triangle, then you will need to take square roots – and we havent said how to do that. Say you want to draw a simple right angled triangle with side lengths in the proportion 3:4:5. To specify the angles would require a trigonometric calculation. We now provide commands for these and some common operations that you might need. You may wonder, how does a computer calculate the value of the sine of an angle, or the square root of a number? The answers to these questions will come later. For now you can just use the following commands without worrying about how the calculation actually happens.

Let us start with square roots. If you want to find the square root of a number `x`, then the command for that is `sqrt`. You simply write `sqrt(x)` in your program and during execution, the square root of `x` will be calculated, and will be used in place of the command. So for example, here is how you can draw an isoceles right angled triangle.

```
forward(100);
left(90);
forward(100);
left(135);
forward(100*sqrt(2));
```

The commands for computing trigonometric ratios are `sine, cosine` and `tangent`. Each of these take a single argument: the angle in degrees. So for example, writing `tangent(45)` will be as good as writing 1.

The commands for inverse trigonometric ratios are `arcsine`, `arccosine` and `arctan`. These will take a single number as an argument and will return an angle (in degrees). For example, `arccosine(0.5)` will be 60 as expected. These commands return the angle in the range -90 to +90. An important additional command is `arctan2`. This needs two arguments, `y` and `x` respectively. Writing `arctan2(y,x)` will return the inverse tangent of `y/x` in the full range, -180 to +180. This can be done by looking at the signs of `y` and `x`, information which would be lost if the argument had simply been `y/x`. Thus `arctan2(1,-1)` would be 135, while `arctan2(-1,1)` would be -45, and `arctan(-1/1)=arctan(1/-1)=arctan(-1)` would also be -45.

Now you will be able to do a triangle with side lengths 300,400,500 as follows.

```
forward(300);
left(90);
forward(400);
left(arctan2(3,-4));
forward(500);
```

As you might guess, we can put expressions into arguments of commands, and put the commands themselves into other expressions and so on.

Some other useful commands that are also provided:

1. `exp, log, log10` : These return respectively for argument x the value of e^x (where e is Euler's number, the base of the natural logarithm), the natural logarithm and the logarithm to base 10.

2. `pow` : This takes 2 arguments, `pow(x,y)` returns x^y.

C++ also has commands `sin, cos, tan` which return the trigonometric ratios given the angle in radians. And for inverse trigonometric ratios we have the commands `asin, acos, atan, atan2` which return the angle in radians.

The name `PI` can be used in your programs to denote π, the ratio of the circumference of a circle to its diameter.

1.6 Concluding Remarks

Although it may not seem like it, in this chapter you have already learned a lot.

First, you have some idea of what a computer program is and how it executes: starting at the top and moving down one statement at a time going towards the bottom. If there are `repeat` statements, the program executes the body of the loop several times; the program is said to *loop* through the body for the required number of iterations.

You have learned the notion of a variable, i.e. a region of memory into which you can read in a value, which can later be used while performing computations.

You have also learned that the language might provide you commands which you can use without having to know how exactly they work. Later on we will see how to ourselves build new commands.

Finally, a very important point concerns observing the *patterns* in whatever you are doing. When we draw a polygon, we repeat the same action several times. This is a pattern that we can mirror in our program by using the `repeat` statement. By using a repeat statement we can keep our program compact; indeed we may be drawing a polygon with 100 sides, but our program only has a few statements. You will see other ways of capturing patterns in your programs later. In general this is a very important idea.

At this point you should also see why the notation used to write programs is called a *language*. A spoken language is very flexible and general. It has a grammatical structure, e.g. there is a subject, verb, and object; or there can be clauses, which can themselves contain subjects, verbs, objects and other clauses. And so long as the structure is respected, you can have many, many, indeed an infinite number of sentences. Similarly, computer programs have a structure, e.g. a `repeat` statement must be followed by a count and a body; but inside the body there can be other statements including a repeat statement. Indeed our treatment of the C++ programming language will be somewhat similar to how you might be taught Tamil or French. Just as language learning is more fun if you read interesting literature, we will introduce the C++ language as we try to solve more and more interesting computational problems. Hope you will find this enjoyable.

It will be important to learn the various grammatical rules of the C++ language as you go along. However, it will also be important for you to develop some intuition for what the rules might be or *ought to be*. For example, we said in the section above that instead of specifying a number for how much to turn, an expression such as `360.0/nsides` can be used. You should ask yourself, is this a general rule? It turns out, indeed, that this is a general rule. In most places where numbers are expected, you may also specify expressions. The computer will evaluate those expressions and use the resulting values. So for example, you may write `cout << nsides*100;` which will cause the perimeter of the polygon being drawn to be printed on the screen (assuming the side length is 100).

The major difficulty in writing programs is not, of course, the grammatical rules of C++. These you will master with some practice. The main difficulty is in deciding what commands to give, what order to give the commands in, how to group them into repeat statements. In this chapter, two ideas have appeared regarding this, and it is worth stating them explicitly.

The first idea is: before you starting to write a program that makes a computer do something, think about how you yourself solve the problem. How would you draw a square? How would you draw a square sitting on top of a real turtle? You will realize that what you write is often just a careful statement of what you yourself would have done. So this has two parts: you should yourself know how to solve the problem, and you should be able to explain in words how you solve the problem. In some sense, programming is a bit like teaching.

The second idea is: whatever activity you are doing, there are likely to be patterns or symmetries in it. If you are drawing a square, then you repeat the same draw-turn pattern four times. Seeing this pattern is extremely important. The pattern contains some insight about the activity which is inherently valuable, but more directly it enables us to write compact programs. Writing compactly and such that the patterns in the activity are reflected in the program is a major goal of programming.

1.7 Overview of the book

The next chapter gives you a bird's eye view of the world of computers and programming. We will study at a high level how computers are constructed, and how real life problems from playing chess to making train reservations can be represented on them. We will try to get an intuitive sense of computer hardware so that the working of a computer as will be discussed in subsequent chapters appears reasonable to you, and you do not feel that there is too much magic in all this.[4]

The subsequent chapters will teach you how to program using C++. The phrase "how to program" can have many interpretations. Here are some of the possible interpretations, starting from the least ambitious going on to the most ambitious:

1. Understanding the different statements in C++. Another way of stating this is: given a program and enough time, can you in principle say what output the program will if you know what input is given to it? The issue here is only *comprehension*, and not design.

2. Ability to write programs to solve problems that you can yourself solve using pencil and paper and given enough time. For example, we know the formula for finding the roots of a quadratic equation. Can we express this computation in a program? Or given the set of marks obtained by students in a course, we would be determine the average mark,

[4] Well, even people who fully understand computers think there is something magical in them, but that is only in a manner of speaking.

or the maximum mark. Can we write a program to do such computation? There are many such problems which you solve routinely without computers, because you have a systematic procedure in mind. Can you express these rules in programs so that the problems can be solved by a computer? If the answer is yes, then you have reached the second level of programming ability.

3. Cooperate with a team of people to solve a large problem. Issues such as how to decompose the problem, how to do your work so that others can use it become important.

4. Discovering new ways of solving a problem. For example, what is the shortest way to go from Mumbai to Tanjore, spending at most Rs 1000? You may think you know many ways to make this travel, however, it is quite likely that you may not know enough to be sure that no better way is possible. Solving such problems requires a higher level of problem solving/programming ability.

The situation is not unlike learning a foreign language. Being able to understand the language is one level of competence. Being able to speak it is the next. Often, your general expression abilities improve as you learn a new language, and you are better able to say things which you didnt know how to say earlier. This is perhaps the analogue of the fourth point mentioned above: ability to solve new problems. Of course, when you learn a new language, you need to learn the etiquette, the way to say things formally, in that culture. This is different from merely being able to converse on the street. It is hoped that you will keep these analogies in mind as you read the book, and gauge your own progress along these directions.

1.7.1 A note regarding the exercises

Programming is not a spectator sport. To really understand programming, you must write many, many programs yourself. That is when you will discover whether you have truly understood what is said in the book. To this end, we have provided many exercises at the end of each chapter, which you should assidously solve.

Another important suggestion: while reading many times you may find yourself asking, "What if we write this program differently". While the author will not be present to answer your questions, there is an easy way to find out – write it differently and run it on your computer! This is the best way to learn.

1.8 Exercises

In all the problems related to drawing, you are expected to identify the patterns/repetitions in what is asked, and use `repeat` statements to write a concise program as possible. You should also avoid excessive movement of the turtle and tracing over what has already been drawn.

1. Modify the program given in the text so that it asks for the side length of the polygon to be drawn in addition to asking for the number of sides.

2. Draw a sequence of 10 squares, one to the left of another.

3. Draw a chessboard, i.e. a square of side length say 80 divided into 64 squares each of sidelength 10.

4. If you draw a polygon with a large number of sides, say 100, then it will look essentially like a circle. In fact this is how circles are drawn: as a many sided polygon. Use this idea to draw the numeral 8 – two circles placed tangentially one above the other.

5. A pentagram is a five pointed star, drawn without lifting the pen. Specifically, let A,B,C,D,E be 5 equidistant points on a circle, then this is the figure A–C–E–B–D–A. Draw this.

6. Draw a seven pointed star in the same spirit as above. Note however that there are more than one possible stars. An easy way to figure out the turning angle: how many times does the turtle turn around itself as it draws?

7. We wrote "360.0" in our program rather than just "360". There is a reason for this which we will discuss later. But you could have some fun figuring it out. Rewrite the program using just "360" and see what happens. A more direct way is to put in statements `cout << 360/11; cout << 360.0/11;` and see what is printed on the screen. This is an important idea: if you are curious about "what would happen if I wrote ... instead of ...?" – you should simply try it out!

8. Read in the lengths of the sides of a triangle and draw the triangle. You will need to know and use trigonometry for solving this.

9. When you hold a set of cards in your hand, you usually arrange them fanned out. Say you start with cards stacked one on top of the other. Then you rotate the ith card from the top by an amount proportional to i (say $10i$ degrees to the left) around the bottom left corner. Now, we can see the top card completely, but the other cards are seen only partially. In particular, only a triangular portion of each card is seen, with the top left corner being at the apex of each triangle. This is the figure that you are to draw. (a) Draw it assuming the cards are transparent. (b) Draw it assuming the cards are opaque. For this some trigonometric calculation will be necessary. In both cases, use `repeat` statements to keep your program small as possible.

10. Draw a pattern consisting of 7 circles of equal radius: one in the center and 6 around it, each outer circle touching the central circle and two others. Try to write a program which minimizes turtle movement. Your program statements should be chosen to exploit the symmetry in the pattern.

Chapter 2

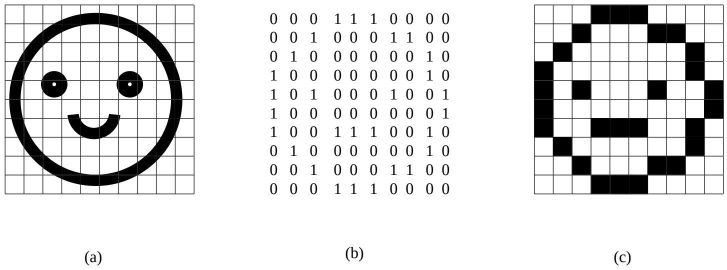
A bird's eye view

At first glance, it is indeed surprising that a single device like a computer should be able to predict the weather, or help design cars, or analyze pictures, or search the books and documents all over the world and tell you where you might find information regarding your topic of interest, or play chess better than the human world champion. The purpose of this chapter is to reduce this surprise somewhat, to make it more believable that a computer could perhaps be able to do all this. The argument has two parts. The first part is the observation that all the problems described above, from predicting the weather to playing chess, can be described in the language of numbers, Mathematics. In a sense, Mathematics is universal, it can be used to analyze almost every phenomenon or process or entity known to man. Once we have reduced the problem we want to solve to some mathematical problem, then what remains is to device a circuit which can solve the problem. Even in this, there is an element of universality: we will argue that a single (but huge) circuit that a modern computer is, is capable of solving many, many problems simply by running appropriate *programs*.

The science of programming, then could be said to be founded on two great sciences. On one side, we have Mathematics, and on the other side, we have the science of designing electrical circuits, Electrical Engineering. We wish to describe the relationship of programming to these sciences. Our description, especially that related to circuit design, will be quite superficial. However, it is hoped that this description will provide some background, a helpful reassurance, as you navigate through subsequent chapters.

We begin with the question of representing real life phenomenon using numbers. We discuss this briefly and we will return to this question in later chapters. The rest of the chapter attempts to provide a *plausible* view of the circuits constituting a computer. We present a very simplified model of a computer and using it explain some basic ideas such as the notion of a program stored in memory, the idea of step by step execution of instructions, and the notion of an *address* for referring to regions of memory. These ideas are central to design of computers as well as programming.

We expect our description of computers in this chapter to be read like a short, popular science article. We will try to anticipate the main questions you might have about how computers work, and answer them at an intuitive level. We will gloss over many details, and also oversimplify several ideas. We cannot give a complete answer to all questions, after all, circuit design and computer architecture are deep subjects, each needing an independent book or more. We believe, however, that you will get a "working knowledge" of computer



```
0 0 0 1 1 1 0 0 0 0
0 0 1 0 0 0 1 1 0 0
0 1 0 0 0 0 0 0 1 0
1 0 0 0 0 0 0 0 1 0
1 0 1 0 0 0 1 0 0 1
1 0 0 0 0 0 0 0 0 1
1 0 0 1 1 1 0 0 1 0
0 1 0 0 0 0 0 0 1 0
0 0 1 0 0 0 1 1 0 0
0 0 0 1 1 1 0 0 0 0
```

Figure 2.1: A picture, its representation, and reconstruction

hardware, providing adequate background for the study of programming to come later.

2.1 Representing real life entities using numbers

Some entities in real life have an obvious mathematical character; any quantity that we can `measure`, such as mass, force, voltage, concentration of chemicals, is naturally expressed numerically. But for some other entities, it is not immediately obvious how they can be expressed using numbers. Can we express pictures or language using numbers? We discuss these questions briefly.

Here is how a picture might be represented using numbers. Consider a black and white picture to begin with. The picture is divided into small squares by putting down a fine grid over it, as in Figure 2.1(a). Then for each small square we determine whether it is more white or more black. If the square is more white we assign it the number 0, if it is mostly black, we assign it the number 1. So if we have divided the picture into $m \times n$ small squares (*pixels*), m along the height and n along the width, we have a sequence of mn numbers, each either 1 or 0 that represents the picture. Figure 2.1 shows the numbers we have assigned to each square. Given the mn number representation, we can reconstruct the picture as follows: wherever a 0 appears, we leave the corresponding square white, wherever a 1 appears, we make the corresponding square black. The reconstruction, using the numbers in Figure 2.1(b) is shown in Figure 2.1(c). As you can see, the reconstructed picture is not identical to the original picture, but reasonably similar. By choosing a finer grid, we would have been able to get a better approximation of the original picture. Since our eye cannot individually see very small squares, it turns out that pixels of size about 0.1 mm are good enough, i.e. the reconstructed picture is hard to distinguish from the original. *Processing* a picture means doing computations involving these numbers. For example, changing every zero to a one and vice versa, will change the picture from "positive" to "negative"!

The idea of putting down a grid over the object of interest is very powerful. Suppose we wish to represent the worldwide weather. Typically, we divide the surface of the globe into small regions. For each region we consider all the parameters relevant to the weather, e.g. the ambient temperature, pressure, humidity. Of course, all points in a region will not have identical temperature, but we nevertheless can choose an approximate representative temperature, if the region is reasonably small. The key to predicting weather are laws of physics: given the current state and knowing the physical characteristics we can say what the next state will likely be. This is a very gross simplification of how the weather is predicted, but, it is correct in essence.[1]

Text can also be represented using numbers. Essentially, we device a suitable code. The most common code is the so called ASCII (American Standard Code for Information Interchange) code. In this, the letter "a" is represented as the number 97, "b" as 98 and so on. Standard symbols and the *space* character also have a code assigned to them. So the word "computer" is represented by the sequence of numbers 99,111,109,112,117,116,101,114. Thus we might be given a sequence representing a paragraph. Finding whether a given word occurs in this paragraph is simply checking whether one sequence of numbers is a subsequence of another sequence of numbers! Once we can express text, we have a foothold for expressing sentences, and all of language! The ASCII code is meant only for the Roman alphabet, there are codes meant for other alphabets, such as the Devanagari alphabet as well.

We will see more real life objects (and mathematical objects too, such as sets, functions) and how to represent them in the rest of the book.

2.2 Representing numbers on a computer

> *God made natural numbers, the rest is the work of man.*
> *– Leopold Kronecker.*

There are, of course, different types of numbers. Starting with the simplest, the so called natural numbers or counting numbers, we have integers (which can be negative), rational numbers, real numbers, and even complex numbers. We begin by discussing how to represent natural numbers. The other types of numbers can in fact be represented using natural numbers. Note that there is some ambiguity about whether 0 is to be considered a natural number; we include it and use the term to mean non-negative integers.

To represent a natural number we first write it in binary. So if we wish to represent the decimal number 25 on a computer, we first write it as 11001 in binary. Now each binary digit, or *bit* of the binary number, is represented separately. For each bit there are only two possibilities: 0 or 1. To represent a bit, we would typically dedicate one capacitor somewhere in our computer, and put a low or high voltage on that capacitor depending upon whether we want to represent a 0 or 1 respectively. If we want to store a 5 bit number, such as 11001, we would have to designate some 5 capacitors, and store respectively high, high, low, low, and high voltages on them. It is of course our reponsibility to keep track of where we store each number, and even which of the capacitors stores the least significant bit, second least significant bit, and so on. High voltage might mean 0.7 volts, low voltage might mean 0 volts.

Using voltages on capacitors to represent numbers is not the only way. It is also possible to designate wires in our computers for this purpose: a low current in the wires might represent 0, and a high current might represent 1 (again with some agreed upon values for what high and low means). Magnetization might also be used: magnetization in one direction might

[1] This is not to say that all physical phenomenon related to the weather are well understood. In fact, many simple things are not understood, e.g. how precisely do rain drops form. However, we understand enough (through the hard work of several scientists) to make predictions with some confidence. All this is of course well outside the scope of this book.

mean 0, in another might mean 1. This is common for storing data on magnetic discs (or tapes, long ago). On optical discs, a reflective region might represent a 1, a non reflective region a 0. An optical CD is divided into a large number of such regions, each of which can be used to store a bit.

You may wonder whether using bits to represent numbers is the only possible way. Why, for instance, do we not use 0 volts to represent 0, 0.1 to represent 1, 0.2 to represent 2, 0.3 to represent 3, and so on. In an ideal world, this idea could be implemented. But in the real world, there are imperfections: it is possible that we intended to raise the voltage of a wire to 0.2 but because of some electrical noise or some circuit imperfection it only got raised to 0.1 This would then change the value that we stored! If this number represents a letter, then it would change a "p" to "q" or something like that – which would be quite unacceptable. On the other hand, suppose we instead have 0.0 volts represent a 0 and 0.7 volts represent a 1. This in effect means: any voltage below 0.35 will be interpreted as 0 and any voltage above 0.35 as 1. Now to cause a misinterpretation we would need to have noise as large as 0.35 volts. This turns out to be unlikely for the technology we have for building such circuits. Because of such considerations, it turns out that using the binary representation is the most convenient.

Once we have found a way to represent numbers using voltages or currents, the task of processing numbers can be expressed as an electrical engineering question: given a certain configuration of voltages or currents, produce another configuration of voltages or currents.

2.2.1 Bits, Bytes, Words

The basic unit of memory is a bit. As mentioned earlier, a bit is typically stored using a capacitor, high voltage across it denoting the value 1 and low voltage denoting the value 0. Groups of 8 bits are called bytes, and a group of 4 bytes constitutes a *word*. We will use this definition of a word, even though it is not as universally as accepted as the definitions of bits and bytes. The term *half-word* is often used to refer to the two bytes constituting each half of a word, and the term *double-word* for two consecutive words of memory.

2.2.2 Storing natural numbers

We have already discussed that natural numbers are stored in binary. To store a number, we need to allocate at least as many bits of memory as there are bits in the number. However, when allocating memory to store a number, we usually do not ask for an arbitrary number of bits; it is customary to ask for a byte, or a half-word, or a word, or a double-word, as will be seen in the following chapters. If we asked for 32 bits, or a word of memory, and wish to store 25 in it, we must first write 25 using 32 bits, as:

$$00000000000000000000000000011001$$

This pattern of bits would then have to be stored in the chosen word.

2.2.3 Storing integers

Integers are different from natural numbers in that they can be negative as well. This throws a new challenge: for example, how would we represent the integer -25?

The simplest representation is the so called sign-magnitude representation. In this, if n bits are used in total, one of these is designated as a *sign bit*. So in addition to the capacitors we dedicate for storing the magnitude, we will also dedicate one capacitor for storing the sign. We could decide that a low voltage on the capacitor indicates that the stored number is positive, while a high voltage indicates that the stored number is negative. We might use the bit in the most significant position as the sign bit, so the representation for -25 using the 32 capacitors would be:

$$10000000000000000000000000011001$$

A more commonly used representation is the so called 2's complement representation. The n bit 2s complement representation is defined as follows. In this the integer x is represented by the binary number x if $0 \leq x \leq 2^{n-1} - 1$, and by the binary number $2^n - x$ if $-2^{n-1} \leq x < 0$.

Thus, in 32 bit 2's complement representation, -25 would be represented by the bit pattern:

$$11111111111111111111111111100111$$

2.2.4 Storing real numbers

Much computing needs to be done with real numbers. For example, velocities of particles, voltages, temperatures and so on in general need not take only integral values. In the scientific world such quantities are typically written using the so called *scientific notation*, in the form: $f \times 10^q$, where the *significand* f typically has a magnitude between 1 and 10, and the *exponent* q is a positive or negative integer. For example the mass of an electron is 9.109382×10^{-31} kilograms, or Avogadro's number is 6.022×10^{23}.

How should we store Avogadro's number on a computer? First of course, we must represent it in binary, this is easily done: it is

$$1.11111110001010101111111 \times 2^{1001110}$$

Note that this is approximate, and correct only to 3 decimal digits. But then, 6.022×10^{23} was only correct to 3 digits anyway. The exponent 1001110 in decimal is 78. Thus the number when written out fully will have 78 bits. We could use 78 bits memory to store the number, however, it seems unnecessary, especially since many of those 78 bits will be 0s anyway. A better alternative, is to store each number in two parts: one part being the significand, and the other being the exponent.

For example, we could use 8 bits to store the exponent, and 24 bits to store the significand, so that a number is neatly fitted into a single word! This turns out to be essentially the method of choice on modern computers. You might ask why use an 8-24 split of the 32 bits and why not 10-22? The answer to this is experience: for many calculations it appears that 24 bits of precision in the significand is adequate, while the exponent size of 8 bits is also needed. There are schemes that use a double word as well and the split here is 11-53, again based on experience.

Note that the significand as well as the exponent can be both positive or negative. One simple way to deal with this is to use a sign-magnitude representation, i.e. dedicate one bit from each field for the sign. Note that we dont need to explicitly store the decimal point (or we should say, binary point!) – it is always after the first bit of the significand. Assuming that the exponent is stored in the more significant part or the word, Avogadro's number would then be stored as:

$$0, 1001110, 0, 11111111000101010111111$$

Two points to be noted: (a) we have put commas after the sign bit of the exponent, the exponent itself, and the sign bit of the significand, only so it is easy to read. There are no commas in memory. (b) Only the most significant 23 bits of the significand are taken. This requires throwing out less significant bits (what happened in this example), but you might even have to pad the significand with 0s if it happens to be smaller than 23 bits.

As another example, consider representing -12.3125. This is -1100.0101 in binary, i.e. 1.1000101×2^3. Noting that our number is negative and our exponent is positive, the representation would be

$$0, 0000011, 1, 11000101000000000000000$$

Again the commas are added only for ease of reading.

The exact format in which real numbers are represented on modern computer hardware and in C++ is the IEEE Floating Point Standard, described in detail in Appendix E. It is much more complicated, but has more features, some of which we will use later.

2.2.5 Storing text

We mentioned earlier that text is represented using the ASCII code: each character is given a numerical code which can be stored on the computer. Each code is an integer in the range 0 through 255 (not all integers correspond to visible characters, though), and hence the code assigned to any character can fit in 1 byte. Indeed, this is perhaps the reason the byte was defined to be made up of 8 bits. In any case, text is represented on a computer by placing the ASCII codes of consecutive letters in consecutive bytes in memory.

2.2.6 Remarks

It is important to note that when stored in memory, all data, be it text or be it numbers, is merely a collection of bits. The same word of memory can be used to store 4 ASCII characters, a natural number, an integer, or a floating point number. The memory simply holds sequences of 0s and 1s, it is upto us to remember what *type* of data we have stored, and thereby correctly interpret the bit sequence.

Another point to be noted is that a word of memory, i.e. 32 bits, can represent 2^{32} distinct bit patterns. So if you decide to use it for storing real numbers (or other kinds of numbers), it can store at most 2^{32} distinct numbers. When we decide to give more space to the exponent, we increase the range of magnitudes we can store, and correspondingly reduce the precision at which each number is expressed. But the number of numbers that can be represented remains at most 2^{32} as before.

But there are infinitely many natural numbers or integers or real numbers! So using just one word of memory (or indeed any fixed number of words), there will always be numbers which we cannot represent. For example, our representation given above cannot represent natural numbers bigger than or equal to 2^{32}. Our integer and real number representations also have similar limitations. In fact, the real number representation is further limited because it is approximate: all numbers are represented only to a finite precision.

By and large these limitations are not serious. But if you feel that for some computation you read much larger numbers or numbers with much higher precision, you can implement them yourself, as is hinted at in Exercise 5.10.

2.3 Organization of a computer

It is customary to think of a computer as consisting of several parts, a memory, an arithmetic logic unit (ALU), input devices such as the keyboard, output devices such as a monitor, and a control unit.[2] The control unit and the ALU are together often called the central processing unit (CPU).

The parts mentioned above are connected together by a network. It is possible to command any part, say the memory, to place data onto the wires constituting the network. Thus the same pattern of voltages that was in the memory appears on the wires. Other components connected to the wires, say the ALU, can be instructed to read this pattern from the wires. This is how data can be said to *flow* between parts of the computer.

In the following sections, we discuss these parts and their operation in greater detail.

2.4 Memory

The memory can be thought of as consisting of a sequence of bytes (which in turn is a sequence of 8 bits). The main memory in a present day computer can be several gigabytes where a gigabyte is 2^{30} bytes.

Suppose that a computer has 2^{30} bytes of memory. It is useful to assume that the bytes in the memory are arranged in a sequence. The yth byte in the sequence is said to have the address y, where $0 \leq y < 2^{30}$. Here is a possible, *word-oriented* organization for this memory.

The memory connects to the rest of the computer using a *data port*, an *address port*, and a *control line*. The data port is a set of 32 wires (corresponding the size of a word) connecting the circuitry of the memory and the rest of the world. The address port is another such set of 30 wires (as many as the logarithm to base 2 of the number of bytes in the memory). The control line is a single wire. The memory circuits allow us to perform two operations on the memory: write a word into it, and read a word from it.

We can write a number x into the word starting at address y by using the following procedure. We begin by placing the number y on the address bus. By this we mean that we convert y to binary, and place a low/high voltage on the ith wire of the address bus as per the ith least significant bit of y. We place the number x, consisting of 32 bits, on the data bus. We place a 0 (i.e. a low voltage) on the control line. All these voltages are held steady for a duration called the *cycle time* of the memory. This causes the circuitry in the memory

[2] Unusual input devices such as temperature or illumination sensors or output devices that control beepers or even motors might also be present.

to do its work, and at the end of the cycle time, we are assured that the value x is deposited in the word starting at address y, i.e. in bytes with addresses $y, y+1, y+2, y+3$. To read the content of the word at y (i.e. the word starting from address y), we place the number y on the address bus, and a 1 (high voltage) on the control line. We again hold this pattern for the duration of the cycle time. If the value z was present in bytes $y, y+1, y+2, y+3$ treated as a single word, then z appears on the data bus. The value on the data wires can then be sensed by the interconnecting circuitry and copied to other parts, e.g. the arithmetic logic unit.[3]

By the way, the phrases "word with address y" or "word at y" or sometimes even just "word y" is used to mean the word starting at address y. Similarly for bytes, half-words, or double-words.

2.4.1 Registers

Sometimes we might have a memory which can store only one word. Such a memory is called a register. Registers can come in useful while building a computer.

2.5 Arithmetic Logic Unit

The arithmetic logic unit (ALU) has circuits using which it is possible to perform arithmetic operations, e.g. addition, subtraction, multiplication, division, comparisons, for numbers in all formats described earlier, natural numbers, integers, and real numbers. Note that a computer cannot directly compute standard functions such as say the sine or cosine. This must be done by performing a sequence of arithmetic operations. We will see this later.

The ALU will also have circuits using which we can convert between different number types, e.g. given a number represented as an integer, what is its representation as a real number (exponent and significand)? Also there will be circuits which can extract bytes or bits out of a word, e.g. when a word containing 4 characters is loaded, and we want to know what the second of those characters is, we must extract just the second byte out of the word.

The ALU also has circuits to remember information such as "was a positive value computed in the last arithmetic operation?". The answer to this question can be either "Yes" or "No". Such values, or equivalently the values "True" or "False" are said to be *Logical Values*. Hence the inclusion of the term *Logic* in the name of this unit. True, false are commonly represented by 1, 0 respectively. When you add two natural numbers, the sum might be become 2^{32} or larger. In that case, the sum cannot be stored in a single word. In this case there is said to have been an *overflow*. The ALU can detect an overflow and remember it. How exactly we can use this information will become clear in Section 2.7.

The set of circuits that seem to be needed might seem bewildering. And indeed a lot of clever design is needed, and it is a science by itself.

For the description to come later, we will consider the ALU to have 2 inputs, input1 and input2, and a single output. The inputs and the outputs will each consist of some number n of wires. For simplicity we will assume $n = 32$, i.e. our ALU operates on 32 bit data. In

[3] Most commonly, it is required that y be a multiple of 4. If all words start on addresses that are multiples of 4, then these do not overlap with each other. This has some advantages, but we will not worry about this.

addition, it will have various control inputs. Each control input can be thought of as a single wire, which if set to 1 will cause the ALU to perform the corresponding function (e.g. add the two numbers on the inputs assuming they are real numbers). There will be auxiliary outputs which will indicate whether the last operation produced a zero result or a positive or negative result and so on.

2.6 Input-Output Devices

The simplest input device is a keyboard. A code number is assigned for each key on the keyboard. When a key is pressed, the corresponding code number is sent to the computer.

The simplest output device could be what used to be called a “dumb” terminal: a screen on which you can merely display essentially the same symbols that you can type from your keyboard. The manner in which you display a symbol is simple: the computer sends the code number associated with the symbol to the screen, and the terminal contains circuitry which causes the corresponding symbol to be displayed at the current cursor position. However, this is a very simplistic description, and more capabilities will be needed even for a dumb terminal. For example, there must be a way for the computer to clear the screen completely.

Modern screens, such as the one you would need to display our turtle, are of course much more complex. You probably know that a computer screen is made up of *pixels* which are arranged in a grid, say 1024 rows and 1024 columns. Associated with each pixel, there is a certain amount of memory, which determines what colour is shown in that pixel. The amount of memory depends on the sophistication of the display. For a simple black and white display, it might be enough to merely decide whether the pixel is to appear white or black. So a single bit of memory is enough. You may also have variations of gray: k bits of memory will be able to store numbers between 0 and $2^k - 1$ and hence that many gray levels. In colour displays we need to simultaneously store the red, green, blue components at each pixel, and so presumably even more bits are needed. Indeed high quality colour displays might use as much as 24 bits of memory for each pixel. To display an image, all we need to do is to store appropriate values in the memory associated with each pixel in the screen. If we have 24 bits of memory per pixel, then because there are $1024 \times 1024 = 2^{20}$ pixels, we will need a memory with addresses between 0 and $2^{20} - 1$, each cell of the memory consisting of 24 bits. A reasonable correspondence is used to relate the pixels and addresses in memory: the colour information for the pixel (i, j) i.e. the pixel in row i and column j (with $0 \leq i, j < 1024$) is stored in address $1024i + j$ of the memory. When the circuitry of the screen needs to display the colour at pixel (i, j) it picks up the colour information from address $1024i + j$ of the memory. When the computer needs to change the image, it merely changes the data in the memory. So to the computer, the screen appears very much like another memory. This memory should not be confused with the main memory of the computer discussed earlier, in which we expect to store data.

Devices such as disks can also be thought of as storing data at certain addresses, however, the addresses no longer refer to specific capacitors in the circuitry, but specific locations on the magnetic surface.

2.7 The Control Unit

As the name implies, the Control Unit controls the other parts of a computer. The control unit can be roughly divided into two parts: an *Instruction-Fetch Unit* (IFU), and a *Decode and Execute Unit* (DEU).

The DEU connects to the other parts of the computer and commands them to perform different actions. For example, the DEU can command the ALU to add the numbers at its inputs. Or the DEU can command the number at the output of the ALU to be moved to the data port of the memory. Following that the DEU may cause some other number to be moved to the address port, and then command the memory to store the data. In fact, the DEU is typically designed so that it can perform different *sequences* of actions. What sequence of actions the DEU will perform will depend upon the *instruction* sent to it by the IFU.

As you might guess, everything in a computer is numerical, and so when we say the IFU sends an instruction to the DEU, what we mean is that a certain number representing what to do next is sent. The circuits in the DEU interpret the numbers it receives from the IFU and decide what parts (e.g. memory, ALU) should be commanded to do what (e.g. send a value from memory to the ALU, or perform addition in the ALU), and in what order.

You may wonder how the IFU knows what numbers to send to the DEU. It merely reads them from the memory! The IFU contains a register, often called the *program counter* (PC) whose function is to hold the address from which the IFU will fetch the next instruction. After fetching that instruction the IFU sends it to the DEU. Then the IFU waits while the DEU does its actions. After the DEU finishes, the IFU starts again. The register PC is designed so that its value can be easily increased[4] Thus after the DEU finishes the value in PC increases so that the instruction following what was just executed can be fetched. This cycle repeats, except in some cases, as we will see below.

To make the ideas more vivid, we present some examples of how instructions might be defined by a computer designer. What we describe is very simplistic and is meant only to help understand the overall mechanism. All this is much more elaborate on real computers. For simplicity we will assume that each instruction sent to the DEU by the IFU consists of two words. We will call the first word the *operation code*, and the second word, the *operand*. The DEU uses the operation code to decide what operation sequence it must perform. The operand is typically interpreted by the DEU as a memory address. Figure 2.2 shows some possible instructions that we could have in a computer.

Suppose as an example, that the PC in the IFU contains the value 100. Suppose further that the word starting at (byte) address 100 contains the value 0, the word following that, i.e. at (byte) address 104 contains the word 90. Suppose the computer starts execution in this state. Here is how the execution would proceed. First, the IFU would attempt to fetch word at the address contained in PC. For this, the content of PC would have to be first moved to the address port of the memory. Then the memory would have to be activated to read the data. Since the address port would receive the value 100, the data at address 100, which we said is 0, would move to the data port. This data would then be sent to the DEU. Note however that we said that each instruction consists of two words. So the program

[4] The value in the PC is *counted up*, hence it is called a counter. Special circuits can be designed to build counters.

counter value would be incremented by 4. We said earlier that the program counter register itself contains circuitry which can do this. So now by a process similar to the one described above, the word at address 104 would be fetched. This would also be sent to the DEU. At this point the IFU would stop and the DEU would take over.

Customarily, the DEU has two registers in which it stores the operation code and the operand it received from the IFU. The DEU uses the operation code to decide what sequence of actions to perform. From Figure 2.2 we know that operation code 0 requires that the data in the operand register be used as an address for the memory, and the word stored at that memory location be moved to ALU input 1. So the DEU accomplishes this by moving the content of the operand register to the address register and so on. After the DEU finishes, the IFU starts executing again. The PC in the IFU is first incremented so that the next instruction can be fetched. The cycle then repeats. Notice that in each cycle the IFU performs the same set of operations; the DEU might behave differently as per the operation code.

Suppose now that we wish to read two natural numbers from the keyboard, and display their sum on the screen. What instructions would we need to send to the DEU? We would begin with the command for reading a natural number from the keyboard: this has operation code 31. Let us arbitrarily decide to store the first natural number in the word starting at address 4000. So we would have to execute an instruction which would be represented by the pair of numbers 31, 4000. Say we decide to use the word starting at 4004 to store the second natural number. The instruction for doing this would be given by the numbers 31, 4004. We would then need to move the numbers to the inputs of the ALU. The operation codes for this are 0 and 1 respectively. So we would need instructions 0,4000 and 1,4004. Then we would need to command the ALU to add. This is done by the instruction 10, 0. After this the result must be placed somewhere, say address 4008 (by which we mean the word starting at address 4008). The instruction for this 2, 4008. Finally we have to print the result on the screen. Printing on the screen has operation code 41, so the instruction we specify is 41,4008. Finally we stop the computer using the instruction 99,0.

So we can put the sequence of numbers 31, 4000, 31, 4004, 0, 4000, 1, 4004, 10, 0, 2, 4008, 41, 4008, 99, 0 into the memory, say starting at address 100, and instruct the IFU to fetch from 100. Then we would accomplish what we wanted: read two numbers and print their sum, and the computer would then stop. Note that the sequence of numbers described above could really be called a program. In fact it is customary to call such a sequence a *machine language program*. We will see later how the C++ programs that you saw in Chapter 1 are related to machine language programs.

2.7.1 Control Flow

Suppose that the IFU is currently sending a certain instruction to the DEU, and suppose the instruction was taken from address x of the memory.[5] Then it is customary to say that the *control* (unit) is at that instruction, or at the address x. The sequences of addresses of the instructions executed by the control unit is said to constitute the *path of control flow*. Alternatively, we might say that *control flows* through that sequence of instructions or

[5] Note that we have said that an instruction consists of 2 words, so saying that the instruction was taken from address x really means that it comes from bytes $x, \ldots, x+7$ of the memory.

Op. Code	Operand	Effect of the instruction
0	x	Move data from address x of memory to input 1 of ALU.
1	x	Move data from address x of memory to input 2 of ALU.
2	x	Move data from ALU output to address x of memory.
10	0	Command the ALU to perform the addition of the values in its inputs, treating them as natural numbers. Store the result in ALU output.
11	0	Command the ALU to perform the addition of the values in its inputs, treating them as integers. Store the result in ALU output.
12	0	Command the ALU to perform the addition of the values in its inputs, treating them as floating point numbers. Store the result in ALU output.
13	0	Command the ALU to treat the inputs as natural numbers and subtract input 2 from input 1, and store the result in ALU output.
14,15	0	Same as above, except that 14 is for integers and 15 for floating point numbers.
30	x	Wait for a key to be pressed on the keyboard. When a key is pressed, the keyboard will send its ASCII value. Store that value in address x of the memory.
31	x	Wait for a natural number to be typed on the keyboard. Receive the value into address x of the memory.
40	x	Send the data stored in address x to the screen. Instruct the screen to interprete data as an ASCII code, and print the corresponding character. So if the data was 97, the character 'a' would be printed.
41	x	Send the data stored in address x to the screen. Instruct the screen to interprete data as a natural number, and print it. So if the data was 97, the characters "97" would be printed.
99	0	Stop.

Figure 2.2: Some possible instructions and their effects

Op. Code	Operand	Effect of the instruction
60	x	Store x into PC.
61	x	Store x into PC if the last result computed by the ALU was 0.
62	x	Store x into PC if the last result computed by the ALU was positive.
63	x	Store x into PC if the last result computed by the ALU was negative.
64	x	Store x into PC if there was an overflow in the last ALU operation.

Figure 2.3: Jump instructions. PC = program counter.

addresses. Note that similar phrases are also used in connection with C++ programs: we will say that the control is at a given statement of the program and so on.

Normally, the control unit executes instructions in the order in which they are stored in the program memory. However, most computers also have instructions that cause the control to *jump* to a different location, i.e. in other words, start fetching and executing instructions from a different address in memory. Figure 2.3 shows examples of jump instructions.

Basically, when a jump instruction is encountered, the DEU writes the operand value into the PC register of the IFU. This way the IFU fetches the next instruction from the newly written value.

It is also possible to jump conditionally. Basically, the DEU writes the operand value only if the specified condition holds. If the specified condition does not hold, then the IFU is not affected. As an example, suppose now that the instruction currently being executed, which came from address y was $61, x$. If the last arithmetic operation produced a 0 value, then the control would jump to address x. If the last arithmetic result was not 0, then the control would continue with executing the next instruction, i.e. the one from address $y + 8$.[6]

Figure 2.4 gives an example of a program which uses jump instructions. It reads two numbers from the keyboard, and prints the second number as many times as the value of the first number.

2.7.2 Some tricky instructions

Before we conclude this section, in Figure 2.5 we introduce a few tricky instructions which we will talk about in later chapters, but also see the exercises. The first 3 use the operand not as the address, but the address of the address. Hence these instructions are said to be *indirect* load, store, and jump respectively.

[6] Remember that each instruction is 2 words, or 8 bytes.

Address	Data	Explanation
196	1	
200	31	Read into address 4000.
204	4000	
208	31	Read into address 4004.
212	4004	
216	0	Move data from 4004 to input 1 of ALU.
220	4004	
224	1	Move data from 196 to input 2 of ALU.
228	196	
232	41	Print data from 4000 to screen.
236	4000	
240	13	Subtract (integers) and move result to ALU output.
244	0	
248	2	Move ALU output to address 4004.
252	4004	
256	62	If last result was positive jump to address 216.
260	216	
264	99	Stop execution.
268	0	

Figure 2.4: Program to print many times

Op. Code	Operand	Effect of the instruction
70	x	Let y be the natural number stored in address x. Use y as an address, and move the data in address y to input 1 of ALU.
71	x	Let y be the natural number stored in address x. Use y as an address, and move the data in the output of the ALU to address y.
80	x	Let y be the natural number stored in address x. Start fetching the instructions from address y.
81	x	Suppose z is the address from which the IFU fetched the current instruction. Store $z + 1$ into address x.

Figure 2.5: Some tricky instructions

2.8 High level programming languages

When the earliest computers were built, they could be used only by writing machine language programs. Indeed, you had to decide where in memory you would store your data, look up the computer manual and determine the operation codes needed to perform the actions you wanted, and then write out the sequence of numbers that would constitute the machine language program. Then these numbers would have to be loaded into the computer memory, and then you could execute the program. As you can see, this whole process is very tiring and error prone.

Fortunately, today, programs can be written in the style seen in Chapter 1. We do not think about what instruction codes to use, nor the address in memory where to store the number of sides of the polygon we wish to draw. Instead, we use familiar mathematical formulae to denote operations we want performed. We give names to regions of memory and store data in them by referring to those names. The computer, of course, really only "understands" instruction codes and memory addresses, and does not understand mathematical notation or the names we give to parts of memory. So how does our nice looking program actually execute on a computer?

Clearly, the nice looking programs we write must first be translated into the language of instruction codes etc. that the computer can understand. This is done by a program called a *compiler*, which fortunately has been written by someone already! The program `s++` that you used in the last chapter is a C++ compiler, which takes a C++ program (e.g. square.cpp) and generates the machine language program (e.g. a.out) which can be directly executed.

Here is a C++ program equivalent to the machine language program we described in the previous section.

```
main_program{
  int num1, num2, num3;
  cin >> num1;
  cin >> num2;
  num3 = num1 + num2;
  cout << num3;
}
```

Can you see the correspondence? The first statement is as discussed in Section 1.3.1, and it defines the variables `num1, num2, num3`. These variables play the role of locations 4000, 4004, 4008 respectively. The statements `cin >> num1;` and `cin >> num2;` do the work done by the instructions 31,4000 and 31,4004. The fourth statement, `num3 = num1 + num2;` is equivalent to executing the instructions 0,4000 followed by 1,4004 followed by 11,4008. Finally, the fifth statement is equivalent to the instruction 41,4008.

A C++ compiler will take a program such as the one above and generate the sequence of instructions equivalent to it, such as the sequence 31, 4000, 31, 4004, 0, 4000, 1, 4004, 11, 4008, 41, 4008, 99, 0 described earlier. Of course, the compiler may not use locations 4000, 4004, 4008 to store the data; there is nothing in the C++ program which says which locations to use. But the compiler will use some three locations, and generate the instruction sequence.

But so long as the sequence does what you want, why would you care which locations are used?

By the way, can you tell why the compiler decided to use the instruction code 11 and not the codes 10 or 12? The compiler can make this decision because of the first statement, which says that the numbers are integers (and not natural numbers or floating point numbers, which would require the codes 10 or 12 respectively to be used).

2.9 Boot Loader

We made a cryptic remark earlier about how the IFU can be "asked to start fetching instructions from location 100". We now explain this.

Most of the memory used in modern computers is said to be *volatile*, i.e. the data stored in it is destroyed when the computer is switched off. However, the memories of most computers contains a non-volatile part, e.g. say the data in addresses 0 to 100 is fixed by the manufacturer and this data stays unchanged no matter how many times the computer is switched on and off. Further, when the computer is switched on, it starts executing a program stored in this non-volatile memory. Typically, this program, often called the *boot loader program*, is only capable of loading the program that the computer is really meant to execute. After loading the real program, the boot loader starts executing the just loaded program.

In the exercises, you are asked to write a boot loader program, which loads a program from the keyboard, and then executes it. This is of course, a difficult and tiring exercise, but you do have all the instructions you need to be able to do it.

When you switch on modern computers, they also have a boot loader which runs. The boot loader loads and runs the operating system, e.g. Linux, or Windows, or Mac OS or whatever. The operating system then communicates with the user and then runs the programs that the user wants. But it all begins with a boot loader!

2.10 A simplified simulator

The programs given with `simplecpp` for this chapter include a simulator for a simplified version of the machine described in this chapter.

The simulator has only 100 words of memory, and each word can only store a two digit number, i.e. values 0 through 99. The simulator only supports the natural number addition and subtraction. The jump instructions are supported.

The simulator must be invoked with a command line argument giving the file to load at start. Sample files `add2numbers.txt` and `printmany.txt` are also included. The file is expected to pairs of numbers, the first giving the address and the second the value. Note that the values and addresses must both be between 0 and 99. For simplicity, the simulator uses word addresses rather than byte address. Further the arithmetic on natural numbers is performed modulo 100.

After loading the file, the program counter, PC, is set to 0. If there is a second command-line argument `fast` then the execution begins immediately and continues till the end. If this argument is absent, then the user must click the mouse for each instruction to execute.

The execution of each instruction is shown in full detail: the register from which data moves is highlighted in blue and the register where it goes is highlighted in red. In case of input/output instructions: the input and output will happen on the terminal window, and not the graphics window.

2.11 Concluding Remarks

The first important point made in this chapter is that for solving any problem, you first need to express it as a problem on numbers.

The second important point was that numbers can be processed using appropriately designed circuits. Such circuits can be controlled using programs, and these programs are sequences of instructions, each of which is also represented using numbers!

Finally, we discussed the correspondence between machine language and C++. In the following chapters we will see more C++ statements, and it is hoped that you will ask yourself how those might get translated to machine code.

It should be noted, however, that the description of computer hardware in this chapter was very simple-minded, and that real hardware is much more elaborate.

2.12 Exercises

1. Write a C++ program equivalent to the machine language program of Figure 2.4.

2. Suppose a certain computer does not have an instruction to multiply. Show that we can perform multiplication of two numbers using add operations and jump operations. For simplicity, your program should simply add the multiplicand to itself as many times as the multiplier.

3. Suppose you want to draw a "+" symbol at the center of a 1024×1024 display. Suppose the display will show a pixel white if you store a 1 at the corresponding memory location. Suppose the "+" is 100 pixels tall and wide, and 2 pixels thick. In which screen memory locations would you store 1s?

4. How many different numbers are represented in the n bit 2's complement representation? Compare this to the sign bit representation discussed in the text, in which we store have 1 bit for storing the sign, and the remaining $n - 1$ bits for storing the magnitude.

5. Is there a bit in the 2s complement representation which can be considered to be a sign bit, i.e. it is 0 for positive numbers and 1 for negative numbers? Having such a bit is convenient because we can quickly tell whether the number is positive or negative.

6. One way to store a rational number p/q is to store p, q separately. Would this be better than performing the division and then storing the resulting real number to a fixed number of bits? What do you think are the tradeoffs?

7. Write a boot loader program. It whould wait and read two integers n and s from the keyboard. The first integer n will give the length (number of words) of the program that will be supplied by the user subsequently, over the keyboard. The second integer s gives the starting address where this program is to be loaded. The boot loader should then read the n additional words from the keyboard and store them into addresses $s, s+1, \ldots, s+n-1$. Finally the boot loader should jump to address s.

   You may need to use some of the instructions we discussed last, the so called tricky instructions.

Chapter 3

Numbers

In this chapter we will see C++ statements for processing numbers. By “processing numbers” we mean actions such as reading numbers from the keyboard, storing them in memory, performing arithmetic or other operations on them, and writing them onto the screen. Clearly, these actions are at the heart of any program. We have already seen examples of these actions in the programs in Chapter 1. In this chapter, we will build up on that and state everything more formally and more generally.

As mentioned in Chapter 2, textual data is also represented numerically on a computer: each character is represented by its numerical ASCII code. Text processing turns out to be a minor variation of numeric processing. Logical data is also represented numerically. We consider these topics briefly, they are considered at length in Section 13.1 and Section 5.7 respectively.

Using the `repeat` statement and what we learn in this chapter we will be able to write some interesting programs. We see some of these at the end.

3.1 Variables and data types

A region of memory allocated for holding a single piece of data (for now a single number), is called a *variable*. C++ allows you to create a variable, i.e. allocate the memory, and give it a name. The name is to be used to refer to the variable in the rest of the program. You have considerable freedom in choosing the names to give to variables, the exact rules are discussed in Section 3.1.3. You have already seen one example of creating a variable, the variable `nsides` of Section 1.3.1. We will see more examples shortly.

When you ask for a variable to be created, you need to specify the type of data you want to store in the variable. By this we mean details such as whether you want to store natural numbers or integers or real numbers. This will determine how numbers will be represented when stored. As we saw in the previous chapter, natural numbers, integers, and real numbers are typically represented in distinct ways. In addition, it is necessary to indicate how much precision you need. If you want to store numbers at high precision, you need a bigger region of memory.

This information, whether the numbers you wish to store are natural numbers, integers, or real, and the amount of of precision, are together said to constitute the *data-type* of the variable, and also of the values stored in the variable. Table 3.1 shows the data types that

Data type	Possible values (Indicative)	Size in bytes (Indicative)	Use
`signed char`	-128 to 127	1	Storing characters or small integers.
`unsigned char`	0 to 255		
`signed short`	-32768 to 32767	2	Storing medium size integers.
`unsigned short`	0 to 65535		
`signed int`	-2147483648 to 2147483647	4	Storing standard size integers.
`unsigned int`	0 to 4294967295		
`signed long`	-2147483648 to 2147483647	4	Storing even longer integers.
`unsigned long`	0 to 4294967295		
`signed long long`	-9223372036854775808 to 9223372036854775807	8	Storing even longer integers.
`unsigned long long`	0 to 18446744073709551615		
`bool`	`false` (0) or `true` (1)	1	Storing logical values.
`float`	Positive or negative. About 7 digits of precision. Magnitude in the range 1.17549×10^{-38} to 3.4028×10^{38}	4	Storing real numbers.
`double`	Positive or negative. About 15 digits of precision. Magnitude in the range 2.22507×10^{-308} to 1.7977×10^{308}	8	Storing high precision and high range real numbers.
`long double`	Positive or negative. About 18 digits of precision. Magnitude in the range 3.3621×10^{-4932} to 1.18973×10^{4932}	12	Storing high precision and very high range real numbers.

Table 3.1: Fundamental data types of C++

are predefined in C++. Once you decide on the data type, and we will say how to do this shortly, you can create a variable by writing the following in your program.

```
data-type variable-name;
```

In this, `data-type` must be a data-type selected from the first column of Table 3.1, and `variable-name` a name chosen as per Section 3.1.3. When this statement is executed a region of (contiguous) memory will be allocated, of size given in column 3 of the table.

As an example, suppose you know that a certain number that you wish to store will be a non-negative integer, with no more than 8 digits. Then to store it, perhaps you would choose `unsigned int` as your data type. Say as an example that the number happens to be a telephone number. Then perhaps you might choose `telephone_number` as the name for your variable, and you would write the following line in your program.

```
unsigned int telephone_number;
```

This would give you a variable consisting of 4 bytes of memory which you could refer to using the name `telephone_number` in the rest of your program. Table 3.1 does not mention the representation scheme to be used for this variable, but from Section 2.2.2 we know that the number would be stored using a 32 bit binary representation.

The phrase *value of a variable* is used to refer to the value stored in the variable. So the stored telephone number (after it is stored, and we will say how to do this) will be the value of the variable `telephone_number`.

We finally note that you can define several variables in a single statement if they have the same type, by writing:

```
data-type variable-name1, variable-name2, ... variable-namek;
```

3.1.1 Remarks on Table 3.1

It should be noted that the size shown for each data-type is only indicative. The C++ language standard only requires that the sizes of `char, short, int, long, long long` to be in non-decreasing order. Likewise, the sizes of `float, double, long double` are also expected to be non-decreasing. The exact sizes are may vary from one compiler to another, and accordingly the possible values that the variables can take will also be different.

The qualifiers `signed` and `unsigned` may be omitted. By themselves, the types, `short, int, long` default to `signed`. The default for `char` may vary from one compiler to another.

Arithmetic on `unsigned` types happens modulo 2^n, where n is the number of bits used for storing the type.

A piece of terminology: the first 9 types in Table 3.1 are said to be `integral` types, and the last 3, `floating` types.

3.1.2 Types `char` and `bool`

The `char` type is most commonly used for storing text. For this use, `char` behaves somewhat differently when it comes to reading data into it (Section 3.1.6) and printing it (Section 3.1.7). However, as you will see, integer arithmetic can be done on `char` data just like the other integer types. So if your programs deals with a large number of integers each having a very small range, then storing them in `char` variables is a fine idea.

The type `bool` is primarily used to store logical values, as will be seen in Section 5.

3.1.3 Identifiers

The technical term for a name in C++ is *identifier*. Identifiers can be used for naming variables, but also other entities as we will see later.

An identifier can consist of letters, digits and the underscore character "_". Identifiers cannot start with a digit, hence you cannot have an identifier such as `3rdcousin`. It is also not considered good practice to use identifiers starting with an underscore for naming ordinary variables. Finally, some words are reserved by C++ for its own use, and these cannot be used as variable names. For example, `int` is a reserved word; it is not allowed to be used as a variable name because it will be confusing. The complete list of reserved words is given in Appendix F.

It is customary to name a variable to indicate the intended purpose of the variable. So if we want to store the velocity in a variable, it is natural to name it `velocity`.

An important point is that case is important in names; so `mathmarks` is considered to be a different name from `MathMarks`. Notice that the latter is easier to read. This way of forming names, in which several words are strung together, and in which the first letter of each word is capitalized, is said to be utilizing camel case, or CamelCase. As you might guess, the capital letters resemble the humps on the back of a camel. There are two kinds of CamelCase: UpperCamelCase in which the first letters of all the words are capitalized, and lowerCamelCase, in which the first letters of all but the first word are capitalized. For ordinary variables, it is more customary to use lowerCamelCase; thus it is suggested that you use `mathMarks` rather than `MathMarks`.

If a variable is important in your program, you should give it a descriptive name, which expresses its use. It is usually best to use complete words, unabbreviated. Thus if you have a variable which contains the temperature, it is better to give it the name `temperature` rather than `t`, or `temp` or `tmprtre`. Sometimes the description that you want to associate with a variable name is very long. Or there is a clarification that the reader should be be aware of. In such cases, it is good to add a comment explaining what you want immediately following the definition, as explained in Section 3.6.

3.1.4 Initializing variables

It is possible to optionally include an initial value along with the definition. So we may write:

```
int p=10239, q;
```

This statement defines 2 variables, of which the first one, `p`, is initialized to 10239. No initial value is specified for `q`, which means that some unknown value will be present in it. The number "10239" as it appears in the code above is said to constitute an integer *literal*, i.e. it is to be interpreted literally as given. Any integer number with or without a sign constitutes an integer literal. All the integer types, including `char` and `bool` must be initialized by specifying an integer literal. However, the following additional ways of specifying integer literals are also provided in C++. For example the words `false` and `true` are literals which stand for the values 0 and 1. So for `bool` variables, it is recommended that you write initializations using these, e.g.

```
bool penIsDown=true;
```

rather than writing `bool penIsDown=1;` which would mean the same thing but would be less suggestive. For convenience in dealing with `char` data, any character enclosed in a pair of single quotes is an integer literal that represents the ASCII value of the enclosed character. Thus you may write

```
char letter_a = 'a';
```

This would indeed store the code, 97, for the letter 'a' in the variable `letter_a`. You could also have written `char letter_a = 97;` but that would not make your intention clear. In general, we may write a character between a pair of single quotes, and that would denote the ASCII value of the character. Characters such as the newline, or the tab, can be denoted by special notation, respectively as '\n' and '\t'. Note that literals such as '\n' and 'a' really represent an integer value. So we can in fact write

```
int q = 'a';
```

This would cause 97 to be stored in the `int` variable `q`.

To initialize floating variables, we need a way to specify real number literals. We can specify real number literals either by writing them out as decimal fractions, or using an analogue of "scientific notation". We simply write an `E` or `e` between the significand and the exponent, without leaving any spaces. Thus we would write Avogadro's number[1], 6.022×10^{23}, as `6.022E23`. The significand as well as the exponent could be specified with a minus sign, if needed, of course. For example the mass of an electron, $9.10938188 \times 10^{-31}$ kg, would be written as `9.10938188E-31`. Thus we may write:

```
float w, y=1.5, avogadro = 6.022E23;
```

This statement defines 3 variables, the second and third are respectively initialized to 1.5 and 6.022×10^{23}. The variable `w` is not initialized.

3.1.5 `const` keyword

Sometimes we wish to define identifiers whose value we do not wish to change. For example, we might be needing Avogadro's number in our program, and it will likely be convenient to refer to it using the name `Avogadro` rather than typing the value everytime. In C++ you can use the keyword `const` before the type to indicate such named constants. Thus you might write

```
const float Avogardro = 6.022E23;
```

Once a name is declared `const`, you cannot change it later. Thus in this case the compiler will complain if you later happen to make an assignment to it.

[1] The number of molecules in a mole of any substance, e.g. number of carbon atoms in 12 gm of carbon.

3.1.6 Reading data into a variable

To read a value into a variable `pqr` we write

```
cin >> pqr;
```

Simply put: when this statement is executed, the computer will wait for us to type a value consistent with the type of `pqr`. That value will then be placed in `pqr`.

The exact execution process for the statement is a bit complicated. First, the statement ignores any *whitespace* characters that you may type before you type in the value consistent with the type of `pqr`. The term whitespace is used to collectively refer to the space character ' ', the tab character '\t', the newline character '\n', the vertical tab '\v', the formfeed character '\f' and the carriage return '\r'. Do not worry if you are unfamiliar with the last three characters in the list. Once you start typing a value consistent with the type of `pqr`, then the appearance of a whitespace character serves as a delimiter. Let us consider an example. Suppose `pqr` has type `int`, then if you execute the above statement, and type

```
      123 56
```

the spaces that you type at the beginning will be ignored, the value 123 will be stored into `pqr`. This is because the space following 123 will serve as a delimiter. The 56 will used in a subsequent read statement, if any. Note further that the value you type will not be received by your program unless you type a newline after typing the value. Thus to place `123` into `pqr` in response to the statement above, you must type a newline either immediately following 123 or following 56.

If `pqr` was of any of the floating types, then a literal of that type would be expected. Thus we could have typed in `6.022e23` or 1.5. If `pqr` was of type `bool` you may only type 0 or 1.

Reading into a `char` variable

You may not perhaps expect what happens when you execute

```
char xyz;
cin >> xyz;
```

In this case the initial whitespaces that you type if any will be ignored, as discussed above. Any non-whitespace value is considered appropriate for the type `char`, so the first such value will be accepted. The ASCII value of the first non-whitespace character that you type will be placed into `xyz`. Note that if you type 1, then `xyz` will become 49. This is because the ASCII value of the character '1' is 49. If you type the letter a, then `xyz` would get the value 97.

Reading several values

If you wish to read values into several variables, you can express it in a single statement.

```
cin >> pqr >> xyz;
```

This is equivalent to writing `cin >> pqr; cin >> xyz;`.

3.1.7 Printing

If you print a variable `rst` of type `bool, short, int` or `long`, writing

```
cout << rst << endl;
```

its value will be printed. A minus sign will be printed if the value is negative. The final `endl` will cause a newline to follow.

If you print a floating type variable, then C++ will print it in what it considers to be the best looking form: as a decimal fraction or in the scientific format.

Printing a `char` variable

Consider the following code.

```
char xyz=97;
cout << xyz << endl;
```

This will cause that character whose ASCII value is in `xyz` to be printed. Thus in this case the letter a will be printed. Following that a newline will be printed, because of the `endl` at the end of the statement.

Printing several values

The two previous statements above can be combined into a single statement if you wish.

```
cout << rst << endl << xyz << endl;
```

3.1.8 Exact representational parameters

Table 3.1 mentions the indicative sizes of the different data types. You can find the exact number used by your compiler by using the `sizeof` command in your program:

```
cout << sizeof(int) << endl;
```

Or `sizeof(double)` and so on as you wish.

You can also determine the largest or smallest (magnitude) representable numbers in the different types. Say for `float`, the expression `numeric_limits<float>::max()` gives the value of the largest floating point number that can be represented. Please do not worry about the complicated syntax of this expression. By using other types instead of `float` or by using `min` instead of `max`, you can get the minimum/maximum values for all types. In order to use this facility, you need to put the following line at the top of your file (before or after other `#include` statements):

```
#include <limits>
```

We will see the exact action of this line later.

3.2 Arithmetic and assignment

We can perform arithmetic on the values stored in variables in a very intuitive manner, almost like we write algebraic expressions. The values resulting from evaluating an arithmetic expression can be stored into a variable by using an assignment statement.

The notion of expressions is similar to that in Algebra. If you have an algebraic expression $x \cdot y + p \cdot q$, its value is obtained by considering the values of the variables x, y, p, q, and performing the operations as per the usual precedence rules. In a similar manner you can write expressions involving C++ variables, and the value of the expression is obtained by similarly considering the values of the variables and performing operations on them, with the same rules of operator precedence. One difference is that often in Algebra the multiplication operator is implicit, i.e. xy means x multiplied by y. In a C++ expression, we need to explicitly write the multiplication operator, which is `*`. All the arithmetic operators `+,-,*,/` are allowed. Multiplication and division have equal precedence, which is higher than that of addition and subtraction which have the same precedence. Some additional operators are also allowed, as will be discussed later. Among operations of the same precedence, the one on the left is performed first, e.g. 5-3+9 will mean 11. We can use brackets to enforce the order we want, e.g. write 5-(3+9) if we want this expression to evaluate to -7. If we had C++ variables `x,y,p,q`, then the expression corresponding to the algebraic expression above would have to be written as `x*y+p*q`. Note that when you use a variable in an expression, it is your responsibility to ensure that the variable has been assigned a value earlier.

An expression causes a sequence of arithmetic operations to be performed, and a value to be computed. However, the computed value is lost unless we do something with it. One possibility is to store the computed value in some variable. This can be done using an assignment statement. The general form of an assignment is:

```
variable = expression;
```

where `variable` is the name of a variable, and `expression` is an expression as described above. Here is an example.

```
int x=2,y=3,p=4,q=5,r;
r = x*y + p*q;
```

This will cause `r` to get the value of the specified expression. Using the values given for the other variables, the expression is simply 2*3+4*5, i.e. 26. Thus `r` will get the value 26.

We could also print out the value of the expression by writing

```
cout << x*y+p*q << endl;
```

Note that when you use a variable in an expression, you must have assigned it a value already, say by initializing it at the time of defines it, or by reading a value into it from the keyboard, or in a previous assignment statement. If this is not done, the variable will still contain some value, only you dont know. If an unknown value is used in a computation, the result will of course be unpredictable in general.

Note that the operator `=` is used somewhat differently in C++ than in mathematics. In mathematics a statement `r = x*y + p*q;` asserts that the left hand side and right hand side are equal. In C++ however, it is a command to evaluate the expression on the right and put the resulting value into the variable named on the left. After the assignment the values of the expressions on either side of the = operator are indeed equal if we consider r on the left hand side to be a trivial expression.

Note however, that we cannot write x*y + p*q = r; because we require the left hand side to be a variable, into which the value of the expression on the right hand side must be stored.

The rule described above makes it perfectly natural to write a statement such as:

```
p = p + 1;
```

This is meaningless in mathematics; in C++ however it just says: evaluate the expression on the right hand side and put the resulting value into the variable named on the left. Assuming p is as in the code fragment given earlier, its value is 4. Thus in this case the value 4+1=5 would be put in p. Note that a statement such as p + 1 = p; is illegal – the left hand side p + 1 does not denote a variable.

3.2.1 Modulo operator: %

In C++, % is the remainder or the modulo operator. Thus the expression m % n evaluates to the remainder of m when divided by n. This operator has the same precedence as *, /.

3.2.2 Subtleties

The assignment statement is somewhat tricky. The first point concerns the floating point representations. Both, float and double are imprecise representations, where the significand is correct to a fixed number of bits. So if an arithmetic operation affects less significant bits, then the operation will have no effect. As an example, consider the following code.

```
float w, y=1.5, avogadro=6.022E23 ;
w = avogadro + y;
```

What is the value of w? Suppose for a moment that we precisely calculate the sum avogadro + y. The sum will be

$$602200000000000000000001.5$$

We will have a problem when we try to store this into a float type variable. This is because a float type variable can only stores significands of 24 bits, or about 7 digits. So in order to store, we would treat everything beyond the most significant 7 digits as 0. If so we would get

$$602200000000000000000000$$

This loss of digits is called *round-off error*. After the round off, this can now fit in a float, because it can be written exactly as 6.022E23. Net effect of the addition: nothing! The variable w gets the value avogadro even though you assigned it the value avogadro + 1.5. This example shows the inherent problem in adding a very small float value to a very large float value.

Some subtleties arise when we perform an arithmetic operation in which the operands have different types, or even simply if you store one type of number into a variable of another type. C++ allows such operations, and could be said to perform such actions reasonably well. However, it is worth knowing what exactly happens.

Suppose we assign an `int` expression to a `float` variable, C++ will first convert the expression into the floating point format. An `int` variable will have 31 bits of precision excluding the sign, whereas a `float` variable only has 24 bits or so. So essentially some precision could be lost. There could be loss of precision also if we assign a `float` expression to an `int` variable. Consider:

```
int x=10;  float y=6.6;
x=y;
```

The value 6.6 is not integral, so C++ tries to do the best it can: it keeps the integer part. At the end, `x` will equal 6. Basically, when a floating value is to be stored into an integer, C++ uses truncation, i.e. the fractional part is dropped. You might want the assigned value to be the closest integer. This you can obtain for yourself by adding 0.5 before the assignment. Thus if you write `x=y+0.5;`, then `x` would become 7, the integer closest to `y`.

When we perform arithmetic operations, it is necessary that the operands be of the same type. If they are, then the result is also computed to be of the same type. If your program asks to perform arithmetic operations on operands of different types, then the operands must be converted so that they have the same type. The rules for this are fairly natural. We always convert less expressive variables to more expressive ones, where `unsigned int` are considered least expressive, `int` more expressive than that and `float` most expressive. If the two variables differ in size, then the smaller is converted to have a larger size. Suppose we have an arithmetic expression `var1 op var2`, where var1 is int and var2 float. Then `var1` will be converted to `float`. If `var1, var2` are `long, int,` then `var2` will be converted to `long`. If the operands are of type `float, long long` then both will be converted to `double`, and so on. After the expression is evaluated, it may either itself form an operand in a bigger expression, or it might have to be stored into a variable. In both cases, there may have to be a further type conversion.

The above discussion leaves open the question: what happens when we write `xyz*100.0` where `xyz` is of type `float`? The key point to note is that a floating literal like 100.0 is by default considered to be of type `double`, and hence the value in `xyz` is first converted to `double` and then the product computed. The product, of course, has type `double`. An integer literal is likewise considered to be of type `int`. You can specify literals of specific types by attaching the suffixes L,F,U which respectively stand for long, float, unsigned. Thus if you write `100LU`, it will be interpreted as a literal of type `long unsigned`, having the value 100.

Here are some simple examples.

```
int x=100, w;
float y,z;
y = 360/x;
z = 360.0/x;
w = 360.0/x;
```

As per the rules stated, `360/x` will be evaluated to have an integer value since both operands are integer. Thus the exact quotient 3.6 will be truncated to give 3. This value will be stored (after conversion to the floating point format) into `y`. In the next statement, `360.0/x` one of the operands is `float`, hence the result will be evaluated as a float, i.e. 3.6. This value will be stored in `z`. In the final statement, the value of the expression will indeed be 3.6, however because `w` is of type `int`, there will have to be a type conversion, and as a result the value stored in `w` will be just 3.

3.2.3 Overflow

For each numerical data type, we have mentioned a certain largest possible and smallest possible value that can be represented. While performing calculations, the results can go outside this allowed range. In this case, what exactly happens is handled differently for different types.

For the `unsigned` data types, the rule is that arithmetic is performed modulo 2^n, where n is the number of bits used. So for example if you add up two `short` numbers, both 65535, then the result will be $(65535+65535) \bmod 65536 = 65534$, where you may note that $2^{16} = 65536$.

For signed integer types, the language does not specify what must happen. In other words, you as a programmer must be careful to ensure that the numbers stay within range.

For the floating point types, `float` and `double`, something quite interesting happens. This is because of the IEEE floating point standard, which is supported in C++. If the result of a computation becomes bigger than the largest representable number for the type, then there is a special bit pattern that gets stored in the variable. This bit pattern behaves like infinity for all subsequent computation. By this, we mean that anything added to infinity remains infinity, and so on (Section E). If you try to print out this pattern, quite likely `inf` would get printed. Thus, you at least get some indication that some overflow occurred during computation.

3.2.4 Explicit type conversion

It is possible to convert an expression `exp` of numerical type `T1` to an expression of type `T2` by writing either

```
T2(exp)
```

or

```
(T2) exp
```

This latter form is a legacy from the C language. The type conversion rules as described earlier apply, e.g. `int(6.4)` would evaluate to the integer value 6.

3.2.5 Assignment expression

It turns out that C++ allows you to write the following code.

```
int x,y,z;
x = y = z = 1;
```

This will end up assigning 1 to all the variables. This has a systematic explanation as follows.

Any assignment, say `z = 1`, is also an expression in C++. Not only is the assignment made, but the expression stands for the value that got assigned. Further, the *associativity* of `=` is right-to-left, i.e. given an expression `x = y = z = 1`, the rightmost assignment operator is evaluated first. This is different from the other operators you have seen so far, such as the arithmetic operators, in which the evaluation order is left to right. Thus, the our statement `x = y = z = 1;` is really to be read as

```
x = (y = (z = 1));
```

Now the expression inside the innermost parentheses, `z = 1` is required to be evaluated first. This not only puts the value 1 into `z`, but itself evaluates to 1. Now the statement effectively becomes

```
x = (y = 1);
```

The execution continues by setting `y` to 1, and then `x` to 1.

3.3 Examples

We consider some simple examples of using the data-types and assignment statements. These do not include the `bool` type which is considered in Section 5.7.

Here is a program that reads in the temperature in Centigrade and prints out the equivalent temperature in Fahrenheit.

```
main_program{
  float centigrade, fahrenheit;

  cout << "Give temperature in Centigrade: ";
  cin >> centigrade;

  fahrenheit = 32.0 + centigrade * 9.0/5.0;
  cout << "Temperature in Fahrenheit: " << fahrenheit << endl;
}
```

Note that the operator `+` is executed last because it has lower precedence than `*` and `/`. The operator `*` executes before `/` because it appears to the left. Note we could have written 9 instead of 9.0. This is because that while multiplying `centigrade`, it would get converted to a float value anyway, since `centigrade` is `float`. Similarly we could have written 5 and 32 instead of 5.0 and 32.0. But what we have written is preferable because it makes it very clear that we are engaging in floating point arithmetic.

In the next program you are expected to type in any lowercase letter, and it prints out the same letter in the upper case.

```
main_program{
  char small, capital;
  cout << "Type in any lower case letter: ";
  cin >> small;

  capital = small + 'A' - 'a';

  cout << capital << endl;
}
```

When the statement `cin >> small;` executes, the ASCII value of the letter typed in by the user is placed in `small`. Suppose as an example that the user typed in the letter q. Then its ASCII value, `'q'` is placed in `small`. This value happens to be 113. To understand the next statement, we need to note an important property of the ASCII codes.

The lower case letters a-z have consecutive ASCII codes. The upper case letters A-Z also have consecutive ASCII codes. From this it follows that for all letters, the difference between the ASCII code of the upper case version and the lower case version is the same. Further, because 'A' and 'a' denote the integers representing the ASCII codes of the respective letters, 'A'-'a' merely gives the numerical difference between the ASCII codes of upper case and lower case of the letter a. But this difference is the same for all letters. Hence given the ASCII code value for any lower case letter, we can add to it 'A' - 'a', and this will give us the ASCII code of the corresponding uppercase letter. So this value gets placed in `capital`, which when printed out displays the actual upper case letter.

To complete the example, note that the ASCII code of 'A' is 65. Thus 'A'-'a' is -32. Since `small` contains 113, capital would get 113 - 32, i.e. 81. This is indeed the ASCII code of Q as required.

3.4 Assignment with `repeat`

What do you think happens on executing the following piece of code?

```
main_program{
  turtleSim();
  int i = 1;
  repeat(10){
    forward(i*10); right(90);
    forward(i*10); right(90);
    i = i + 1;
  }
  wait(5);
}
```

Imagine that you are the computer and execute the code one statement at a time. Write down the values of different variables as you go along, and draw the lines traced by the turtle as it moves. You will probably be able to figure out by executing 2-3 iterations. It is strongly recommended that you do this before reading the explanation given next.

In the first iteration of the `repeat`, `i` will have the value 1, and this value will increase by 1 at the end of each iteration. The turtle goes forward `10*i`, i.e. a larger distance in each iteration. As you will see, the turtle will trace a “spiral” made of straight lines.

We next see another common but important interaction of the assignment statement and the repeat statement. Consider the following problem. We want to read some numbers, from the keyboard, and print their average. For this, we need to first find their sum. This can be done as follows.

```
main_program{
  int count;
  cout << "How many numbers: ";
  cin >> count;

  float num,sum=0;
  repeat(count){
    cout << "Give the next number: ";
    cin >> num;
    sum = sum + num;
  }

  cout << "Average is: ";
  cout << sum/count;
  cout << endl;
}
```

The statement sum = sum + num; is executed in each iteration, and before it is executed, the next number has been read into num. Thus in each iteration the number read is added into sum. Thus in the end sum will indeed contain the sum of all the numbers given by the user.

3.4.1 Programming Idioms

There are two important programming idioms used in the programs of the previous section.

The first idiom is what we might call the *sequence generation idiom.* Note the value of the variable i in the first program. It started off as 1, and then became 2, then 3, and so on. As you can see, by changing the starting value for i and adding a different number to i inside the loop instead of 1, we could make i take the values of any arithmetic sequence (Exercise 5). You will find this idiom helpful in solving many problems.

The second idiom is what we might call the *accumulation idiom.* This was seen in the second program. The variable sum was initialized to zero, and then the number read in each iteration was added to the variable sum. The variable sum was thus used to *accumulate* the values read in each iteration. Stating this differently, suppose the number of numbers read is n, and suppose the values read were $v_1, \ldots, v_n$. Then after the execution of the loop in the second program the variable sum has the value:

$$0 + v_1 + v_2 + \cdots + v_n$$

Here we have written 0+ explicitly to emphasize that the value calculated actually also depends on the value to which sum was initialized, and that happened to be zero, but it is a choice we made.

You might wonder whether this idea only works for addition or might work for other operators as well. For example C++ has the command `max`, where `max(a,b)` gives the maximum of the values of the expressions `a,b`. Will using `max` help us compute the value of the maximum of the values read? In other words, what would happen if we defined a variable `maximum` and wrote

```
maximum = max(maximum, num);
```

instead of `sum = sum + num;`? For simplicity, assuming $n = 4$ and also assuming that `maximum` is initialized to 0 just as `sum` was, the value taken by `maximum` at the end of the repeat will be:

$$\max(\max(\max(\max(0, v_1), v_2), v_3), v_4)$$

Will this return the maximum of v_1, v_2, v_3, v_4? As you can see this will happen only if at least one of the numbers is positive. If all numbers are negative, then this will return 0, which is not the maximum. Before we abandon this approach as useless, note that we actually have a choice in deciding how to initialize `maximum`. Clearly, we should initialize it to as small a number as possible, so that the values v_i cannot be even smaller. We know from Section 3.1.8 that it suffices to choose `-numerical_limits<float>::max()`. Thus our initialization becomes:

```
maximum = - numerical_limits<float>::max();
```

which we put in place of the statement `sum=0;` in the program.

There is another way to do this also, which you might find simpler. We could merely read the first value of `num`, and assign `maximum` to that. Thus the program just to calculate the maximum of a sequence of numbers will be as follows. Note that we now repeat only `count-1` times, because we read one number earlier.

```
main_program{
  int count;
  cout << "How many numbers: ";
  cin >> count;

  float num,maximum;

  cout << "Give the next number: ";
  cin >> maximum;

  repeat(count-1){
    cout << "Give the next number: ";
    cin >> num;
    maximum = max(maximum,num);
  }
  cout << "Maximum is: " << maximum << endl;
}
```

This program does not behave identically to the program sketched earlier, i.e. obtained by initializing `maximum` to `- numerical_limits<float>::max()`. The exercises ask you to say when the programs might differ, and which one you prefer under what circumstances.

3.4.2 Combining sequence generation and accumulation

Often we need to combine the sequence generation and accumulation idioms.

Suppose we want to compute n factorial, written as $n!$, which is just short hand for the product $n \times (n-1) \times (n-2) \times \cdots \times 2 \times 1$. How can we do this?

The key point to note is that we merely need to take the product of the sequence of numbers $1, 2, \ldots, n-1, n$, and this is a sequence that we can generate. But if we can generate a sequence, then we can easily take its product, i.e. accumulate it using the multiplication operator.

```
main_program{
  int n, facn=1, i=1;
  cin >> n;

  repeat(n){
    facn = facn * i;
    i = i + 1;
  }
  cout << facn << endl;
}
```

In the above program, if you ignore the statement `facn = facn * i;`, then we merely have our sequence generation program, with the sequence 1 to `n` being generated in the values taken by the variable `i`. However, the value generated in each iteration is being multiplied into the variable `facn` which was initialized to 1. Hence in the end the variable `facn` contains the product of the numbers from 1 to `n`, i.e. `n!` as we wanted.

But the idea of building up does not stop here. Here is e, the base of the natural logarithm expressed as an infinite series.

$$e = 1 + \frac{1}{1!} + \frac{1}{2!} + \frac{1}{3!} + \cdots$$

Can we compute the sum of the first n terms of this? This is useful, because it gives a good approximate value of e. We can do it by adding just one more layer to our previous program.

```
main_program{
  int n, facn=1, i=1;
  double e=1.0;
  cin >> n;

  repeat(n){
    facn = facn * i;
    e = e + 1.0/facn;
    i = i + 1;
  }
  cout << e << endl;
}
```

Observe how little this program differs from the previous one. Indeed, we are calculating $n!$ in this program as well, and so the code for that is still present. But we are adding the reciprocal of $i!$ calculated in each iteration to `e`, that is the extra code.

In the above code we have declared the variable `facn` to be of type `int`, just so that you can see the correspondence with the previous program. However, $n!$ grows very fast and even 16! is larger than what can be accommodated in an `int`. Fortunately, while computing e, we dont need $n!$ to be represented exactly. So it is better to define `facn` as a `double`, which can accommodate $n!$ even for very values of n, albeit approximately.

The exercises require you to combine these idioms in other ways. However, the basic idea is simple. If you want to compute a certain quantity which is the sum/product or in general an accumulation of some sequence, ask how would you just generate that sequence. If you can generate the sequence, you can accumulate it. The terms of the sequence might themselves be accumulations. So you ask how you could generate the sequence whose accumulation will give you the term. And so on. Sometimes you will have to generate and combine multiple sequences. Sometimes the sequences will come from the keyboard. However the general idea remains the same: to accumulate a sequence first generate it.

Summing series is an important computation, explored at more length in Chapter 7.

3.5 Some operators inspired by the idioms

Because sequence generation and accumulation occur commonly in code, C++ includes operators that can be used to express these idioms compactly.

3.5.1 Increment and decrement operators

A key statement in the sequence generation idiom is `i=i+1;`. This tends to occur quite frequently in C++ programs. So a short form has been provided. In general you may write

```
C++;
```

which merely means `C = C + 1;`, where `C` is any variable. This usage is very useful.

For completeness, we describe some additional, possibly confusing, feature of the `++` operator. Turns out that for a variable `C`, `C++` is also an expression. It stands for the value that `C` had before 1 was added to it. Thus if you wrote

```
int x=2,y;
y = x++;
```

after execution `y` would be 2 and `x` would be 3. The operator ++ written after a variable is said to be the (unary) *post-increment* operator. We recommend that you avoid a statement such as `y=x++;` and instead write it as the less confusing `y=x; x++;`. It is worth noting that in the modern era programming is often done by teams, and so your code will be read by others. So it is good to write in a manner that is easy to understand quickly.

You may also write `++C`, which is the unary *pre-increment* operator operating on `C`. This also causes `C` to increase by 1. `++C` also has a value as an expression: except the value is the new value of `C`. Thus if you wrote

```
int x=2,y;
y = ++x;
```

both x,y would get the value 3. Again this will usually be better written as ++x;y=x; because it is less confusing.

Likewise, C--; means C = C - 1;. This is also a very useful operator, and is called the post decrement operator. As an expression C-- has the value that C had before the decrementation. Analogously, you have the pre-decrement operator with all similar properties. Again, it is recommended that you use the expression forms sparingly.

3.5.2 Compound assignment operators

The accumulation idiom commonly needs the statement vname = vname + expr;, where vname is a variable name, and expr an expression. This can be written in short as:

```
vname += expr;
```

The phrase vname += expr is also an expression and has as its value the value that got assigned. Analogously C++ has operators *=, and -=, /=. These operators are collectively called the compound assignment operators.

The expression forms of the operator += and others are also quite cryptic and hence confusing. It is recommended that you use these expression forms sparingly.

3.6 Comments

Text beginning with two slashes // and going all the way to the end of line is said to constitute a *comment*. Likewise text starting with the characters /* and ending with */. A comment is not meant to be executed. Its purpose is to put in explanatory or auxiliary information (e.g. when was the program written and by whom) along with the code. By writing comments, you can explain why you have written the program as you have, so that other programmers (or you yourself rereading after some time during which you might forget) can understand your code. Use the first format if you want to put in a short comment, and the second format if you want a long comment.

Here is an example of how comments might be added to the last program of Section 3.4.1.

```
/*********************************************************************
Program to read in numbers and print their maximum.  The number of
numbers whose maximum is to be printed is given first, and then the
numbers are given.

Author: Abhiram Ranade
Date: Aug 3, 2012
*********************************************************************/
main_program{
  int count;  // the number of numbers whose max is to be printed.
  cout << "How many numbers: ";
  cin >> count;

  float maximum; // keeps track of the maximum found so far.
  cout << "Give the next number: ";
  cin >> maximum;

  float num;     // for reading in subsequent numbers

  repeat(count-1){ // we already read one number into maximum, so we
                   // need to read only count-1 more.
    cout << "Give the next number: ";
    cin >> num;
    maximum = max(maximum,num);
  }

  cout << "Maximum is: " << maximum << endl;
}
```

As you can see, the program begins with a comment giving what the program does and the name of the author and date of writing the program. This is a long comment, and so the it is contained in `/*` and `*/`.

The other comments are of two kinds. First, we have comments that describe the purpose of each variable defined in the program. Such comments are very useful. Of course, we could give such good names that then the comments would not be needed at all. For example, instead of using the name just `maximum`, we could have used `maximum_value_read_so_far`. In this case, the comment would not have been necessary.

The second kind of comment explains code. Every step doesn't have to be explained. For example, it is quite unnecessary to write a comment `// read in next value` after the line `cin >> num;`, especially since it is preceded by the line asking for the next value. However, it is not entirely obvious why we wrote `repeat(count-1){ ... }` rather than the more natural `repeat(count){ ... }`. So this is explained by writing a comment.

Writing good comments is an art. The comments should not repeat what is obvious from reading the code. If the code itself makes the motivation clear, that is indeed the best. In that case leave it alone!

3.7 Invariants

It is customary to explain what happens in a loop by writing something called an *invariant*. The word invariant means "something that does not change", like the conservation laws in Physics, e.g. the total energy of the system is the same before and after the event. In the context of programming, the term has come to mean just a formal assertion about the values taken by the variables when control arrives at a specific statement in a program.[2]

[2] In Theorem **??** of Section **??** we will see an invariant stated in the form of a conservation law: the values of certain variables `Large` and `Small` may change but their GCD, the greatest common divisor does not change.

Such invariants are most useful in connection with loops, but they may we written for other statements in the program as well.

For example, consider the program for calculating the value of e from Section 3.4.2. We can make the following observations about the values of the different variables as control enters the loop for the tth time, where t goes from 1 to `n`.

1. The variable `facn` will have the value $t - 1!$. Note that $0!$ is defined to be 1.

2. The variable `e` will equal the sum of the first t terms of the series

   $$\frac{1}{0!} + \frac{1}{1!} + \frac{1}{2!} + \ldots$$

   i.e. the sum of terms ending with the term $1/(t-1)!$. Note here that $0! = 1$.

3. The variable `i` will have the value t.

These observations constitute the "invariants that hold when the loop is entered for the tth time".

When we wrote the program, we perhaps didnt state these invariants explicitly, but we intuitively did have them in mind. But intuition is not enough. It is important to write down the invariants, because there is a tendency to get confused about questions such as "Does the program add up all the terms including $1/n!$ or only till $1/(n-1)!$?" So to be clear about such questions, it is best to clearly state the invariants.

You might wonder whether we can cross check, or prove whether our invariants we have written down are correct, after all it is possible to get confused when many variables and their values at different times are involved. Yes, we can in fact prove the correctness of invariants.

The basic idea is to use mathematical induction. To make the argument clearer, it is good to define some notation. Let us use i_t, fac_t, e_t to respectively denote the values of the variables `i, fac, e` on the tth entry into the loop. The invariant we want to prove can be written as:

$$fac_t = (t-1)!, \quad e_t = \frac{1}{0!} + \ldots + \frac{1}{(t-1)!}, \quad i_t = t \tag{3.1}$$

For the base case, we consider $t = 1$, i.e. the values on the first entry. By inspection of the code we can see that these values are

$$fac_1 = 1 = 0!, \quad e_1 = 1 = \frac{1}{0!}, \quad i_1 = 1$$

Thus we have proved the invariant for $t = 1$. So we will now assume that the invariant is true on the tth entry and argue that it must also be true on the $(t+1)$th entry. So we will assume that equation (3.1) holds, and we must prove:

$$i_{t+1} = t+1, \quad fac_{t+1} = t!, \quad e_t = \frac{1}{0!} + \ldots + \frac{1}{t!}$$

To prove this, let us examine what happens during the tth iteration. First we execute `facn = facn * i;` at this point `facn` and `i` have the values that they had on entry, i.e.

$facn_t = (t-1)!$ and $i_t = t$. Thus the value assigned to facn is $(t-1)! * t = t!$. The variable facn does not change subsequently in this iteration, hence this is the value of facn on the $t+1$th entry to the loop. Thus we have proved that $facn_{t+1} = t!$ as we wanted. The second statement executed in the loop after the t entry is `e = e + 1.0/facn;`. The variable e has not changed so far in this iteration, hence its value is the same as on entry, i.e. $1/0!+\ldots+1/(t-1)!$. But facn has acquired a new value, $t!$, as we just argued. Thus after the execution of the statement, the value of e will be $1/0!+\ldots+1/(t-1)!+1/t! = 1/0!+\ldots+1/t!$. This value will not change till the entry to the loop again, i.e. for the $t+1$th time. Thus $e_{t+1} = 1/0! + \ldots + 1/t!$. But this is exactly what we wanted proved. Finally the third statement sets i to i+1. But when control arrives at the third statement in the tth iteration, its value continues to be what it was at the beginning, i.e. t. Thus the assignment `i = i + 1` causes the value $t+1$ to be assigned to i. But this value does not change till the beginning of the next iteration. Thus we have $i_{t+1} = t+1$. But this is also what we wanted to prove. Thus our induction is complete, and we have proved the invariant.

The present loop is simple enough that our invariants might seem "obvious". However, for more complicated loops, a proof will be needed, and in that case we must use induction, just as we did above.

Having written down the invariants, there need be no confusion about which terms the program adds up. The invariant says that on entry to the nth and last iteration the variable e would have the sum till the term $1/(n-1)!$. But during the nth iteration, the term $1/n!$ would get added, and hence in the end e will have the sum till $1/n!$.

It is a good idea to write down the main invariants as comments in the code.

3.8 Blocks and variable definitions

It turns out that most C++ programmers would write the average computation program from Section 3.4 slightly differently, as follows.

```
main_program{
  int count;
  cout << "How many numbers: ";
  cin >> count;

  float sum=0;
  repeat(count){
    cout << "Give the next number: ";
    float num;                         // defined inside the loop
    cin >> num;
    sum = sum + num;
  }

  cout << "Average is: ";
  cout << sum/count;
  cout << endl;
}
```

As you can see, the main difference is that the variable `num` is defined inside the loop rather than outside. We first explain how the variable definition is executed in the new program. As you might guess, the variable indeed gets created when control reaches the definition statement. From the time of creation, the variable is available to the program, *until the time the control reaches the end of the loop body in the current iteration.* In other words, the variable is *destroyed* when the control reaches the end of the body! Thus, in each iteration of the loop, the variable is created and destroyed. Of course, *destroying* a variable is only notional, the computer merely assumes that the memory that was given is now available for use. It should also be noted that the variable cannot be used outside the repeat loop, or before its definition inside the loop.

Experienced programmers prefer to write the average computation code in the new style, because in this the definition of `num` is placed close to its use. Placing definitions close to the use makes it easier to read the program, especially if it has many variables and the loop bodies are large.

Next we will state the general rules for defining variables. First we need the notion of a *block*.

3.8.1 Block

The region of the program from an opening brace, {, to the corresponding closing brace, }, is called a *block*. Thus the entire program forms a block, and the body of a repeat also forms a block, which is contained inside the block consisting of the entire program. If there is a `repeat` inside a `repeat`, then the block corresponding to the body of the former is contained inside the block associated with the latter. As you can see, two blocks must either be completely disjoint, or one of them must be completely contained in the other. It is also useful to define the *parent block* of a variable definition: it is the innnermost block in which the variable is defined.

3.8.2 General principle 1

Now, we can restate more formally what we stated earlier. When control reaches a variable definition, the corresponding variable is created. The variable is destroyed when the control leaves the parent block of the definition. The variable is potentially available for use in the region of the program starting at the point of its definition, and going to the end of its parent block. It is also convenient to say that a variable is *live* from the time it is created to the time it is destroyed.

We have already discussed how this principle applies to the variable `num` of the program given above.

The principle also applies to the variable `sum` in the program. Its parent block is the main program itself, and indeed, the entire portion of the program from the point of its definition to the end of the program can refer to the variable `sum`.

3.8.3 General principle 2

The matter of giving names to variables, is somewhat similar to the way in which we give names to human beings.

Let us first discuss how we name human beings. Ideally, you might think that we should insist that all human beings be given different names. But of course, this does not happen. It is perfectly possible that there exist two families in Mumbai both of which name their son Raju. In that case whenever a reference is made to "Raju" in either family, it is deemed to refer to the son in that family. There is no confusion. Notice however, that usually the same name is not given to two children in the same family.

As another example, consider the name Manmohan. In most families in India, the name would be considered to be referring to the Prime Minister of India. Suppose now that a certain family decides to name their son "Manmohan". In this family, after the birth of the son, if anyone speaks of Manmohan, it would probably be considered as referring to the son. You could say that the son "overshadows" the Prime Minister in this family.

Variable naming in C++ is almost as flexible as naming of human beings, including the idea of shadowing. The analogue of the family is a block of the program.

In a C++ program, it is possible to use the same name in two distinct variable definitions, provided the definitions have different parent blocks. Each variable will be created when control reaches the definition, and will be destroyed when control reaches the end of the block containing the definition. If at any time instant, only one variable with a given name is live, then references to that name will be considered to refer to that variable. However, if two variables are live with the same name, then more recently defined variable will shadow the variable defined earlier, i.e. references to the name will be considered to refer to the recently defined variable.

Here is an example of a program in which there are two variables with the name `num`, but they are not live at the same time.

```
main_program{
  int sum=0;
  repeat(5){
    int num;
    cin >> num;
    sum += num;
  }
  cout << sum << endl;
  int prod=1;
  repeat(5){
    int num;
    cin >> num;
    prod *= num;
  }
  cout << prod << endl;
}
```

In this case the references to `num` in the first loop must refer to the variable created as a result of the definition in the first loop. Similarly the references in the second. This is what you would intuitively expect, and indeed the program will compute the sum of the first 5 numbers that it reads, and the product of the next 5.

Here is an example of a program in which there is shadowing.

```
main_program{
  int p=10;
  repeat(3){
    cout << p << endl;
    int p=5;
    cout << p << endl;
  }
  cout << p << endl;
}
```

In this program, when the control arrives at the inner definition, `int p=5;`, a new variable will be created, and all subsequent references to the name `p` till the end of the body of the `repeat` statement will be considered to refer to this new variable, due to shadowing. Thus the second print statement will refer to the new variable, and will thus cause 5 to be printed. Notice that when the first and the third print statement is executed, only the variable created by the first definition is live, and thus they will cause 10 to be printed. Thus this code when executed will cause the sequence of numbers 10, 5, 10, 5, 10, 5, 10 to be printed.

3.9 Concluding remarks

The first step in computing with numbers is to reserve space in memory for the number. Statements which do this are called *definitions*. A definition reserves the space and also gives it a name. The reserved space, together with its name, is said to constitute a *variable*, and the data stored in the variable is said to be the value of the variable. Of course, what is stored in memory is always a sequence of bits. How we interpret the bits depends upon the *type* of the variable, As discussed in Chapter 2, the same pattern of 32 bits might mean one value for a variable of type `unsigned in`, another for a variable of type `int`, and yet another for a variable of type `float`.

When we refer to the name of a variable in a program, we almost always refer to the *value* stored in the variable, except when the name appears on the left side of an assignment statement, when it refers to the memory associated with the variable, i.e. whatever is the value on the right hand side is to be stored in this memory. Perhaps this observation is useful to prevent being confused by statements such as `p = p + 1;` which are incorrect in mathematics but which have are meaningful in computer programs. We noted that the assignment statement is somewhat subtle, because of issues such as rounding, and converting between different types of numbers.

We also saw two important programming idioms: sequence generation, and accumulation. These will come up in the exercises and in later chapters.

As we discussed, it is important that you have a very good idea of what you will use each variable for. You should go over your program to make sure that you are using it for the claimed purpose and the claimed purpose alone. We also discussed the notion of loop invariants. It is good to write down the main loop invariants as comments in your program.

3.10 Exercises

1. What is the value of `x` after the following statements are executed? (a) `x=22/7;` (b) `x=22.0/7;` (c) `x=6.022E23 + 1 - 6.022E23` (d) `x=6.022E23 - 6.022E23 + 1` (e) `x=6.022E23 * 6.022E23`. Answer for three cases, when `x` is defined to be of type `int`, `float`, `double`. Put these statements in a program, execute and check your conclusions. You may notice that `inf` is printed in one case – this is short for infinity and happens when the number in consideration has become bigger than what C++ can represent in the given type.

2. For what values of `a,b,c` will the expressions `a+(b+c)` and `(a+b)+c` will evaluate to different values?

3. I want to compute the value of $\binom{100}{6} = \frac{100\times99\times98\times97\times96\times95}{1\times2\times3\times4\times5\times6}$. I have many choices in performing this computation. I can choose the order in which to perform the multiplications and divisions, and I can choose the data type I use for representing the final and intermediate results. Here is a program which does it in several ways. Guess which of these are likely to give the correct answer, nearly the correct answer, or the wrong answer. Then run the program and check which of your guesses are correct.

```
main(){
  int x = 100 * 99 * 98 * 97 * 96 * 95/ (1 * 2 * 3 * 4 * 5 * 6);
  int y = 100/1 * 99/2 * 98/3 * 97/4 * 96/5 * 95/6;
  int z = 100/6 * 99/5 * 98/4 * 97/3 * 96/2 * 95/1;

  int u = 100.0 * 99 * 98 * 97 * 96 * 95/ (1 * 2 * 3 * 4 * 5 * 6);
  int v = 100.0/1 * 99/2 * 98/3 * 97/4 * 96/5 * 95/6;
  int w = 100.0/6 * 99/5 * 98/4 * 97/3 * 96/2 * 95/1;

  cout << x << " " << y << " " << z << endl;
  cout << u << " " << v << " " << w << endl;
}
```

4. What is the state of the computer, i.e. what are the values of the different variables and what is on the screen, after 4 iterations of the loop of the spiral drawing program of Section 3.3? Write down your answer without running the program. Then modify the program so that it prints the values after each iteration and also waits a few seconds so you can see what it has drawn at that point. Run the modified program and check whether what you wrote down earlier is correct.

5. Write a program that prints the arithmetic sequence $a, a + d, a + 2d, \ldots, a + nd$. Take a, d, n as input.

6. Write a program that prints out the geometric sequence $a, ar, ar^2, \ldots, ar^n$, taking a, r, n as input.

7. Write a program which reads in side, nsquares, q. It should draw nsquares as many squares, all with the same center. The sidelength should increase by q starting at side. Repeat with the modification that the sidelength should increase by a factor q.

8. Write a program which prints out the squares of numbers from 11 to 99.

9. What does the following program draw:

```
main(){
  turtlesim();
  int i=0;

  repeat(30){
    left(90);
    forward(200*sine(i*10));
    forward(-200*sine(i*10));
    right(90);
    forward(10);
    i++;
  }
}
```

10. The ASCII codes for the digits 0 through 9 are 48 through 57. Suppose in the third statement below, the user types in two digits. The ASCII codes for the digits will then be placed in p,q. You are to fill in the blanks in the code such that dig1 gets the value of the digit in p (not the value of its ASCII code), and similarly dig2 should get the value of the digit in q. Finally, the integer n should contain the value of the number in which p is in the tens place and q in the units place.

```
char p,q;
int dig1,dig2,n;
cin >> p >> q;   // equivalent to cin >> p; cin >> q;
dig1 = ...
dig2 = ...
n = ...
```

For example, if the user typed '1','2', then p,q will contain the values 49,50. At the end we would like dig1,dig2,n to be respectively 1,2,12.

11. Write a program that takes as input the coordinates of two points in the plane and prints out the distance between them.

12. Write the program for computing the maximum of numbers as suggested initially in Section 3.4.1, i.e. the one in which maximum was to be initialized to the value (-numerical_limits<float>::max()). Does this program behave identically (i.e. give the same result for the same inputs) to the program given at the end of the Section 3.4.1? If you think the programs behave differently, state the inputs for which the programs will behave differently.

13. Write a program to approximately compute e^x by adding first 15 terms of the series

$$e^x = 1 + \frac{x^1}{1!} + \frac{x^2}{2!} + \frac{x^3}{3!} + \ldots$$

State the important invariants for the loops in your program.

14. Write a program that computes the value of an nth degree polynomial $A(x) = a_0 + a_1x + a_2x^2 + \ldots + a_nx^n$. Assume that you are given n then the value x, and then the coefficients $a_0, a_1, \ldots, a_n$. State the important invariants for the loops in your program.

15. Evaluate the polynomial, but this time assume that you are given the coefficients in the order $a_n, a_{n-1}, \ldots, a_0$.

16. What does the following program compute?

```
double x;
int n;
cin >> n;
repeat(n){
  x = x * x;
}
```

17. Draw a smooth spiral. The spiral should wind around itself in a parallel manner, i.e. there should be a certain point called "center" such that if you draw a line going out from it, the spiral should intersect it at equal distances as it winds around. State the important invariants for the loops in your program.

18. What will be the effect of executing the following code fragment?

```
float f1, f2,centigrade=100;
f1 = centigrade*9/5 + 32;
f2 = 32 + 9/5*centigrade;
cout << f1 << ' ' << f2 << endl;

char x = 'a', y;
y = x + 1;
cout << y << ' ' << x + 1 << endl;
```

Chapter 4

Simplecpp graphics

The graphics commands we introduced in Chapter 1 are quite limited. For one thing, wouldnt it be nice to have several turtles that move or even draw lines simultaneously? It would also be nice, if we could have other shapes, rather than just triangles, that move on the screen. For example, if we want to create an animation of bouncing balls, it would be nice to have moving circles rather than triangles. Many of these things are supported in `simplecpp`, as we will see in this chapter.

4.1 `initCanvas` and `closeCanvas`

To access the more general graphics facilities, it is more convenient to use the command:

```
initCanvas();
```

This opens a window, but does not create a turtle at its center. Commands `canvas_width()`, `canvas_height()` return the width and height of the canvas in pixels. You may also invoke the command as

```
initCanvas(name,w,h)
```

where `name` is a quoted string meant to be the name given for the canvas, and `w,h` should indicate the width and height you desire for the canvas window. To remove the window, use

```
closeCanvas();
```

instead of `closeTurtleSim();`.

A coordinate system is associated with the canvas window. You may find it slightly unusual: the origin is at the top left corner, and x coordinates increase rightward, and y coordinates downward. The coordinates are measured in pixels. Note however that internally, `simplecpp` considers the coordinates of objects to be real numbers of type `double`. These real coordinates are converted to integers only when needed for the purpose of displaying the objects.

4.2 Multiple Turtles

We can create multiple turtles very easily, by writing:

```
Turtle t1,t2,t3;
```

This will create 3 turtles, respectively named `t1, t2, t3` at the center of the window created using `initCanvas()`. Yes, the turtles will all be at the center, stacked one on top of the other. We next see how we get them untangled.

The basic idea is: any command you used in Chapter 1 to affect the turtle will also work with these turtles, but you must say which turtle you are affecting. For this, you must write the command following the name of the turtle, the two joined together by a dot: ".". Thus, to move turtle `t1` forward by 100 steps, we merely write:

```
t1.forward(100);
```

Likewise, to turn `t2` we would write

```
t2.left(90);
```

The same thing applies to other commands such as `right, penUp, penDown`.

Here is a program which will use 3 turtles to draw 3 octagons, aligned at 120 degrees to each other.

```
main_program{
  initCanvas();
  Turtle t1, t2, t3;

  t2.left(120);
  t3.left(240);

  repeat(8){
    t1.forward(100);
    t2.forward(100);
    t3.forward(100);

    t1.left(360.0/8);
    t2.left(360.0/8);
    t3.left(360.0/8);
  }
  wait(5);
}
```

4.3 Other shapes besides turtles

Three other shapes are allowed besides turtles: circles and axis-parallel rectangles, and straight line segments. Text is also considered to be a kind of shape.

4.3.1 Circles

Circles can be created by writing:

```
Circle c1(cx,cy,r);
```

Here, `cx,cy,r` must be numerical expressions which indicate the radius of the circle, and the x and y coordinates of its center. The created circle is named `c1`.

4.3.2 Rectangles

An axis parallel rectangle is defined as follows

```
Rectangle r1(cx,cy,Lx,Ly);
```

where `cx,cy` should give the coordinates of the center, and `Lx,Ly` the width and height respectively. The created rectangle has name `r1`.

4.3.3 Lines

A line segment can be defined as:

```
Line line1(x1,y1,x2,y2);
```

This creates a line named `line1` where `x1,y1` are the coordinates of one endpoint, and `x2,y2` the coordinates of the other.

4.3.4 Text

If we want to write text on the screen, it is also considered a kind of shape. The command

```
Text t1(x,y,message);
```

in which `x,y` are numbers and `message` is a text string can be used to write the message on the screen. So you might use the command `Text txt(100,200,"C++");` to write the text `C++` on the screen centered at the position (100,200). Another form is:

```
Text t2(x,y,number);
```

Here `number` can be a numerical expression. The value of the expression at the time of execution of this statement will comprise the text. It will be centered at the coordinates `(x,y)`.

4.4 Commands allowed on shapes

Each shape mentioned above can be made to move forward and rotate, and it has a pen at its center which can be either up or down.

In addition, for any shape `s`, we have the commands

```
s.moveTo(x,y);
s.move(dx,dy);
```

where the former moves the shape to coordinates `(x,y)` on the screen, and the latter displaces the shapes by `(dx,dy)` from its current position.

You can change the size of a shape also. Every object maintains a scale factor, which is initially set to 1, based on which its size is displayed.

```
s.scale(relfactor);
s.setScale(factor);
```

Here `relfactor, factor` are expected to be `double`. The first version multiplies the current scale factor by the specified `relfactor`, the second version sets the scale factor to `factor`.

You can rotate shapes except `Rectangle` and `Text` using the `left` and `right` commands as for the shape `Turtle`. Later on we will see the `Polygon` shape, which can be rotated.

Suppose `s` is a shape. Then the following command causes an image of the shape to be printed on the canvas, at the current position of `s`.

```
s.imprint();
```

After this, the shape might move away, but the image stays permanently. You can print as many images of a single shape as you desire. The new image overwrites older images, if any.

You can also decide whether a shape `s` is to appear in outline, or it is to be filled with some color. For the former, use the command

```
s.setFill(v);
```

where `v` must evaluate to `true` or false. This command does not apply to `Line` shapes. The color can be specified by writing:

```
s.setColor(color);
```

where `color` is specified for example, as `COLOR("red")`. Instead of red, other standard color names, e.g. blue, green, yellow, white, black can be used. Use all lowercase letters. Alternatively, you may specify the color by giving intensities of 3 primary colors, red, green, blue respectively.

```
s.setColor(COLOR(redVal, greenVal, blueVal));
```

The 3 values should be numbers between 0 and 255. As you may guess, red and blue together give purple, while red and green give yellow.

You may hide or unhide a shape `s` using the commands

```
s.hide();
s.show();
```

respectively.

4.4.1 Resetting a shape

For each shape except `Turtle`, an `reset` command is provided. This command takes the same arguments as required for creation, and recreates the shape using the new values. For example, you could have

```
Circle c(100,100,15);
wait(1);
c.reset(100,100,20);
```

This would have the effect of expanding the circle.

4.5 Clicking on the canvas

The command `getClick()` can be used to wait for the user to click on the canvas. It causes the program to wait until the user clicks. Suppose the user clicks at a point `(x,y)` on the screen. Then the value `65536*x+y` is returned by the command. Note that the click is considered to be happening on some pixel, i.e. the coordinates `x, y` of the click position are integers. The value returned by `getClick()` is also of type `int`.

As an example, the following program waits for the user to click, and then prints out the coordinates of the point at which the user clicked.

```
main_program{
  int clickPos;

  initCanvas();

  clickPos = getClick();

  cout << "Click position: ("<< clickPos/65536 <<", "
       << clickPos % 65536 <<")\n";
}
```

4.6 Projectile Motion

We will now write a program that simulates the motion of a projectile. Suppose that the projectile has initial velocity 1 pixel per step in the x direction, and -5 pixels per step in the y direction (note that the y coordinate grows downward, so this is upward velocity). Suppose gravitational acceleration is 0.1 pixels per step^2. For simplicity assume that the velocity only changes at the end of each step: at the end of each step 0.1 gets added to the y component velocity.

```
#include <simplecpp>

main_program{
  initCanvas("Projectile motion", 500,500);
  int start = getClick();

  Circle projectile(start /65536, start % 65536, 5);
  projectile.penDown();

  double vx=1,vy=-5;

  repeat(100){
    projectile.move(vx,vy);
    vy += .1;
    wait(0.1);
  }
  wait(10);
}
```

The program waits for the user to click. It then places a projectile, a `Circle` at the click position. Then it moves the projectile as per its velocity. The pen of the projectile is put down so that the path traced by it is also seen. The projectile is moved for 100 steps.

4.7 Best fit straight line

Suppose you are given a set of points $(x_1, y_1), (x_2, y_2), \ldots, (x_n, y_n)$. Your goal is to find a line $y = mx + c$ which is the closest to these points. We will see a way to do this assuming a specific definition of "closest".

A natural definition of the distance from a point to a line is the perpendicular distance. Instead, we will consider the "vertical" distance $y_i - mx_i - c$. We will try to minimize the total distance of all points from our line; actually, since the quantity $y_i - mx_i - c$ can be positive or negative, we will instead minimize the sum of the squares of these quantities, i.e.

$$\min \sum_{i=1}^{n} (y_i - mx_i - c)^2$$

Basically we have to select m, c such that the above quantity is minimized. At the chosen value of m, the above sum must be smallest, i.e. the derivative of the sum must be 0.

$$0 = \frac{\partial}{\partial m} \sum_{i=1}^{n} (y_i - mx_i - c)^2 = -2 \sum_{i=1}^{n} x_i (y_i - mx_i - c)$$

The terms of this can be rearranged as an equation in m and c:

$$m \sum_i x_i^2 + c \sum_i x_i = \sum_i x_i y_i$$

We can get another equation by asserting that the quantity to be minimized become as small as possible for the chosen value of c, or at that value the derivative must become 0:

$$0 = \frac{\partial}{\partial c} \sum_{i=1}^{n} (y_i - mx_i - c)^2 = -2 \sum_{i=1}^{n} (y_i - mx_i - c)$$

This can also be rewritten as an equation.

$$m\sum_i x_i + nc = \sum_i y_i$$

Define $p = \sum_i x_i^2$, $q = \sum_i x_i$, $r = \sum_i x_i y_i$, and $s = \sum_i y_i$. Then our equations are $pm+qc = r$ and $qm + nc = s$. These equations are easily solved symbolically, giving

$$m = \frac{nr - qs}{np - q^2} \qquad c = \frac{ps - qr}{np - q^2}$$

The program is given below. The variable names in it are as per the discussion above.

```
main_program{
  cout << "Number of points: ";
  int n; cin >> n;   // number of points to which the line is to be fit.

  initCanvas("Fitting a line to data",500,500);

  double p=0, q=0, r=0, s=0;
  Circle pt(0,0,0);  // create a dummy circle for use later
  repeat(n){
    int cPos = getClick();
    double x = cPos/65536;
    double y = cPos % 65536;
    pt.reset(x,y,5);
    pt.imprint();

    p += x*x;
    q += x;
    r += x*y;
    s += y;
  }
  double m = (n*r - q*s)/(n*p - q*q);
  double c = (p*s - q*r)/(n*p - q*q);
  Line l(0,c, 500, 500*m+c);
  wait(10);
}
```

4.8 Concluding Remarks

If it appears to you that defining shapes is like defining variables, you would be right! Indeed, statement such as:

```
Circle c1(100,100,10), c2(300,200,15);
```

does indeed define two variables, c1 and c2. The commands discussed above are invoked on these variables, and as a result they cause the images on the screen to be changed. But from the view of the C++ compiler, c1, c2 are in fact variables.

Just as ordinary variables can be defined inside repeat loops, so can these shapes. But just as ordinary variables will get destroyed once we get to the end of the parent block, so will these shapes.

Further, the names of the shapes, Circle, Rectangle, Line, Turtle in fact are the data types of the corresponding variables. These are special data types created for simplecpp. C++ allows creation of data types such as these. We will study this in Chapter 14. For now, you can just use them.

4.9 Exercises

1. Draw an 8 × 8 chessboard having red and blue squares. Hint: Use the imprint command. Use the repeat statement properly so that your program is compact.

2. Plot the graph of $y = \sin(x)$ for x ranging in the interval 0 to 4π. Draw the axes and mark the axes at appropriate points, e.g. multiples of $\pi/2$ for the x axis, and multiples of 0.25 for the y axis.

3. Modify the projectile motion program so that the velocity is given by a second click. For example, the velocity should be taken to be proportional to the distance between the projectile (first click) and the second click. Or you can require the second click to be the highest point reached by the projectile as it moves. For this you may note that if u_x, u_y are the initial velocities of the projectile in the x, y directions, and g the gravitational acceleration, then maximum height reached is $\frac{u_y^2}{2g}$. The horizontal distance covered by the time the maximum height is reached is $\frac{u_x u_y}{g}$.

4. Modify the projectile motion program to trace the trajectories of the projectile for the same initial velocity and different angles. At what angle does the projectile go farthest?

5. Write a program to produce the following effect. First a square appears on the screen. Then a tiny circle appears at the center. Slowly, the circle grows until it touches the sides of the circle. Then both the circle and the square start shrinking until they vanish.

6. Suppose you are given some observed positions of a projectile. Each position is an (x, y) pair. You are further told that the projectile is surely known to pass through the origin (0,0). Derive the best fit trajectory for the given points, such that it passes through (0,0).

7. Write a program that accepts 3 points on the canvas (given by clicking) and then draws a circle through those 3 points.

8. Write a program that accepts 3 points, say p, q, r. Then the program draws the line joining p, q. Then the line is rotated around the point r, slowly, through one full

rotation. The key question here is how to rotate a line through a point which is not its center. This can be done in two ways. You could calculate the next position of the line, and then `reset` the line to that position. Alternatively, you can observe that a rotation about an external point such as r can be expressed as a rotation about the center and a translation, i.e. a `move`. This will require you to calculate the amount of translation.

9. Write a program that accepts numbers d, r where r is the radius of a circle, and d the distance of point P from the center of the circle. Think of the circle as a mirror, and P as a light source. You are to draw a "ray diagram" for this configuration. Light rays emanate from P, and if they reach the circle, will get reflected by the infinitesmal mirror at the point of contact, i.e. the angle made by the incident ray with the radius at the point of contact will be equal to the angle made by the reflected ray.

   Clearly, the rays that will be reflected will be those emerging between the common tangets from P to the circle. Read in an additional number n from the user. Pick $n-1$ rays that divide the angle between the common tangents into n equal parts. These will clearly reflected off the circle. Draw these rays as well as the reflected rays. Extend the reflected rays backwards so that they meet the line joining P and the circle center. The points where the rays meet this line can be said to be the image of the light source in the mirror; as you will see this will not be a single point, but the image will be diffused. This is the so called spherical aberration in a circular mirror.

Chapter 5

Conditional Execution

Suppose we want to calculate the income tax for an individual. The actual rules of income tax calculation are quite complex. Let us consider very simplified rules as follows:

> For males, there is no tax if your income is at most Rs. 1,80,000. If your income is between Rs. 180,000 and Rs. 500,000 then you pay 10% of the amount by which your income exceeds Rs. 180,000. If your income is between Rs. 500,000 and Rs. 800,000, then you pay Rs. 32,000 plus 20% of the amount by which your income exceeds Rs. 500,000. If your income exceeds Rs. 800,000, then you pay Rs. 92,000 plus 30% of the amount by which your income exceeds Rs. 800,000.

In the programs that we have written so far, each statement was executed once, or each statement was executed a certain number of times, as a part of a repeat block. The statements that we have learned do not allow us to express something like "If some condition holds, then execute a certain statement, otherwise execute some other statement.". This *conditional* execution is required for the tax calculation above.

The main statement which expresses conditional execution is the `if` statement. We will also discuss the `switch` statement, which is sometimes more convenient. We also discuss logical data, and how it can be stored in the `bool` type.

5.1 The `If` statement

We first give the program which calculates the tax, and then explain each statement.

```
main_program{
  float income; // in rupees.
  float tax;    // in rupees.

  cout << "What is your income in rupees? ";
  cin >> income;

  if(income <= 180000) tax = 0;                 // first if statement
  if((income >  180000) && (income <= 500000))  // second if statement
    tax = (income - 180000)* 0.1;
```

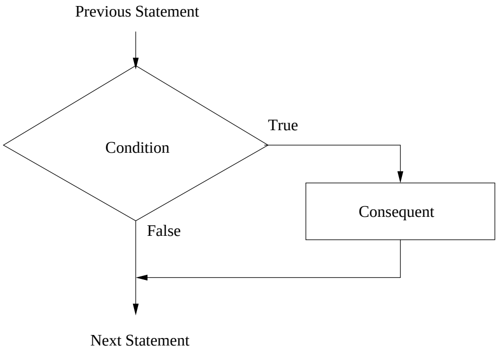


Figure 5.1: Flowchart for simple `if` statement

```
  if((income >  500000) && (income <= 800000))  // third if statement
    tax = 32000+(income - 500000)* 0.2;
  if(income > 800000)                            // fourth if statement
    tax = 92000+(income - 800000)* 0.3;

  cout << "Tax is: " << tax << endl;
}
```

This program uses the simple form of the `if` statement, which is as follows.

```
if (condition) consequent
```

In this the `condition` must be an expression which evaluates to `true` or `false`. We will soon describe how such expressions can be written. In any case, the execution of the `if` statement begins with the evaluation of the `condition` expression. If it evaluates to `true`, then the `consequent`, which can be any C++ statement, is executed. If the `condition` evaluates to `false`, then the `consequent` is ignored. At this point the execution of the `if` statement ends, and control passes to the next statement in the program. Pictorially, this is often shown in the form of a *flowchart*, Figure 5.1. In this figure, boxes are used to hold statements to be executed, or actions to be performed. It is customary to write conditions inside diamonds. Lines join the boxes and diamonds showing how control can flow. As you can see, after evaluating the `condition`, either the true branch is taken, in which case the `consequent` is executed, or the false branch is taken in which case the control directly goes to the next statement.

The simplest form of `condition` is as follows.

```
exp1 relop exp2
```

where `exp1` and `exp2` are numerical expressions, and `relop` is a relational operator, e.g. `<,>,<=,>=,==,!=` which respectively stand for less than, greater than, less than or equal, greater than or equal, equal, and not equal. Thus in the first `if` statement in the program, `income <= 180000` is a condition. If during execution, the value of the variable `income` is at most 180000, then the condition evaluates to `true`, and the `condition` is said to *succeed*. If so the `consequent` is executed. Thus `tax` is set to 0. If `income` is greater than 180000, the condition evaluates to false, and is said to *fail.* In this case the consequent is not executed, i.e. tax remains unchanged. Similarly, in the last if statement, the condition is income > 800000. The consequent here, tax = 92000 + (income - 800000) * 0.3 is executed if and only if the value of income is greater than 800000.

It is possible to specify a more complex condition in the if statement. For example, you may wish to perform a certain operation only if some two conditions are both true. In other words, you want condition1 to be true *and* condition2 to be true. Thus our condition can be a conjunction (*and*) of two or more conditions. This is written as follows.

```
condition1 && condition2 && ... && conditionn
```

The characters && should be read as "and".[1] In our second if statement, we have an example of this. Here, the compound condition is true only if both the subconditions, income > 180000 and income <= 500000 are true. In other words, the compound condition is true only if the income is between 180000 (exclusive) and 500000 (inclusive). Only in this case is the tax set to (income - 180000)* 0.1, i.e. 10% of the amount by which the income exceeds 180000.

Note that we can have a compound condition which holds if at least one of some set of conditions holds. Such a condition is said to be a disjunction of (sub) conditions and is expressed as:

```
condition1 || condition2 || ... || conditionn
```

The characters ||, constitute the logical *or* operator, i.e. the compound condition is true if condition1 is true or condition2 is true and so on. Finally, one condition can be the negation of another condition, written as follows:

```
!condition
```

where the condition !condition is said to be the negation of condition. The condition !condition is true if condition is itself false, and !condition is false if condition is true.

We note that the second if statement can also be written as:

```
  if(!((income <=  180000) || (income > 500000)))
    tax = (income - 180000)* 0.1;
```

Notice that (income <= 180000) || (income > 500000) is true if income is either less than or equal to 180000 or greater than 500000, i.e. if the income is *not* in the range 180000 (exclusive) and 500000 (inclusive). But the ! at the beginning negates this condition, so the entire condition is true only if the income is indeed in the range 180000 (inclusive and 500000 (exclusive). But this is the same condition as tested in the second if statement in the program!

It is important to clearly understand how the above program is executed. The execution is as usual, top to bottom. After printing out a message and reading the value of income, the program executes the first if statement. For this the condition in it is checked, and then the consequent is executed if the condition is true. After this the second if statement is executed. So every if statement will be executed; the conditions have been so designed

[1] The single character & is also an operator, but it means something different, see Appendix G.

so that the condition in only one if statements will evaluate to true, and hence only one consequent statement will be executed. Perhaps the way in which control flows is more obvious in the flowchart of the entire program, shown in Figure 5.2. Note that once we discover a certain condition to be true, e.g. that the income is at most 180000, we know that the other conditions cannot be true. So the natural question arises: why should we even check them?

The more general if statement, discussed in the next section, allows you to prevent such unnecessary checks. But before discussing that, we discuss the notion of *blocks*.

5.2 Blocks

In the if statement discussed above, the consequent was expected to be a single statement. In general, we might want to execute several statements if a certain condition held, not just one. The block construct helps us in this case.

As discussed earlier, a block is simply a collection of statements that are grouped together in braces, { and }. By putting statements into a block, we are making a single compound statement out of them. A block can be placed wherever a single C++ statement is required, e.g. as the consequent part of the if statement. Suppose for example, we want to print a message "This person is in the highest tax bracket." if the income is more than 8 lakhs, as well as calculate the tax, we would replace the fourth if statement in the program with the following.

```
if (income > 800000){
    tax = 92000+(income - 800000)* 0.3;
    cout << "This person is in the highest tax bracket." << endl;
}
```

You have already used a block as a part of the repeat statement. Let us now note that the general form of the repeat statement is:

```
repeat (count) action
```

where action is any statement including a block. Thus we may write

```
repeat (10) cout << "Test." << endl;
```

which will cause the message "Test." to be printed 10 times.

5.3 Other forms of the if statement

The if-else statement has the following form.

```
if (condition) consequent
else alternate
```

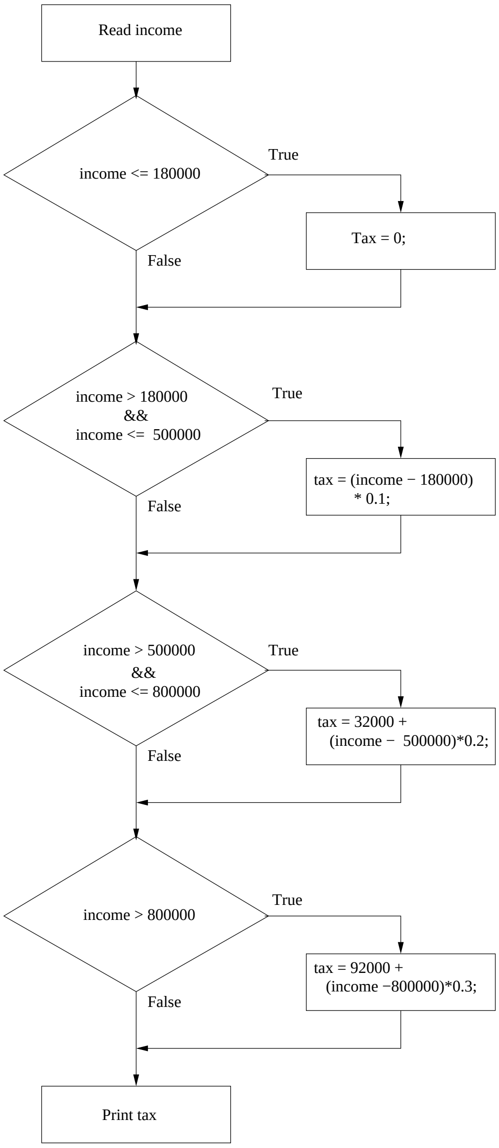


Figure 5.2: Flow chart for the first income tax program

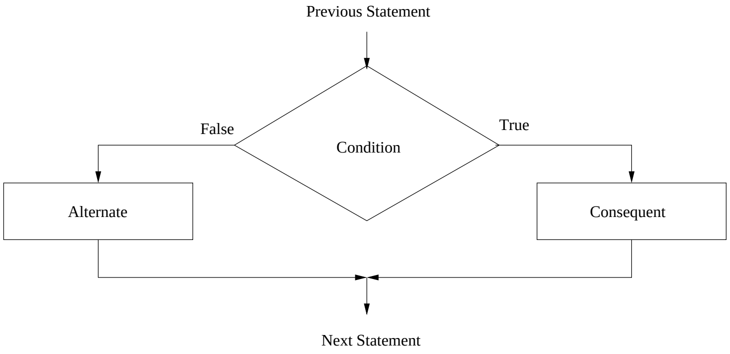


Figure 5.3: If statement with else clause

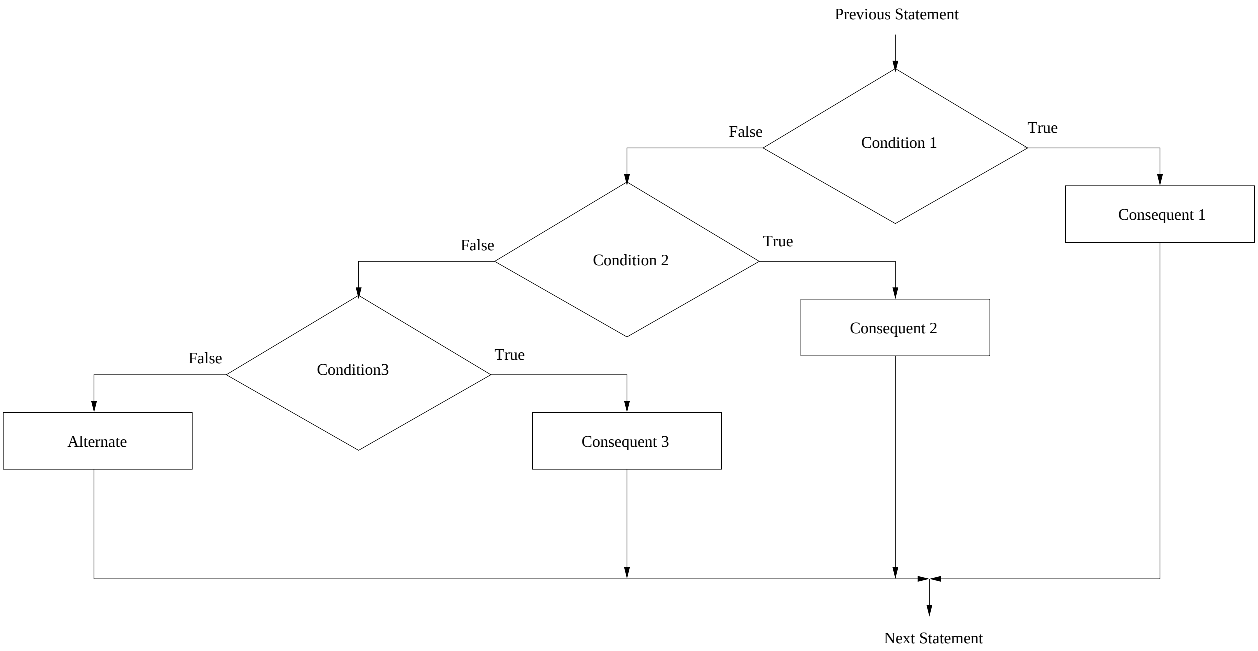


Figure 5.4: Most general if, with 3 conditions

This statement also begins with the evaluation of `condition`. If it is true, then as before `consequent` is executed. If the `condition` is false, however, then the `alternate` statement is executed. So exactly one out of the statements `consequent` and `alternate` is executed, depending upon whether the `condition` is true or false. This is shown pictorially in the flowchart of Figure 5.3.

The most complex form of the `if` statement is as follows.

```
if (condition1) consequent1
else if (condition2) consequent2
else if (condition3) consequent3
...
else if (conditionn) consequentn
else alternate
```

This statement is executed as follows. First, `condition1` is checked. If it is true, then `consequent1` is executed, and that completes the execution of the statement. If `condition1` is false, then `condition2` is checked. If it is true, then `consequent2` is executed, and that completes the execution of the statement. In general, `condition1, condition2, ...` are executed in order, until some `conditioni` is found to be true. If so, then `consequenti` is executed, and the execution of the statement ends. If no condition is found true, then the `alternate` is executed. It is acceptable to omit the last line, i.e. `else alternate`. If the last line is omitted, then nothing is executed if none of the conditions are found true. Figure 5.4 shows a flowchart, for 3 conditions.

Now we can rewrite our tax calculation program as follows.

```
main_program{
  float income, tax;

  cout << "What is your income? ";
  cin >> income;

  if(income <= 180000) tax = 0;       // new first if
  else if(income <= 500000)           // new second if
    tax = (income - 180000)* 0.1;
  else if(income <= 800000)           // new third if
    tax = 32000+(income - 500000)* 0.2;
  else
    tax = 92000+(income - 800000)* 0.3;


  cout << "Tax is: " << tax << endl;
}
```

Notice that this program contains only 3 conditions, rather than 4 as in the previous program. This is because if all the 3 conditions are false, we know that the income must be bigger than 800000. Thus even without checking this condition we can directly set the tax to `92000+(income - 800000)* 0.3`.

Also note that the second and third conditions are much simpler! In the first program, we checked if the income was larger than 180000 and at most 500000. In the new program, we know that the "new second if" is executed only if the condition of "new first if" failed, i.e. if the income was greater than 180000. But then, we dont need to check this again in the "new second if". So it suffices to just check if income is at most 50000. The third `if` statement also simplifies similarly.

Further note that the original program would check each of its 4 conditions no matter which one is true, whereas in this program as soon as the first true condition is found, the corresponding consequent action is performed, and the subsequent conditions are not checked. Thus the new program is more efficient than the previous program. Figure 5.5 shows the flowchart for the new program. By comparing to Figure 5.2, perhaps it is easier to appreciate how much different the new program is.

5.4 A different turtle controller

The turtle driving programs we saw in chapter 1 required us to put information about the figure we wanted to draw right into program, i.e., the exact sequence of forward and turn commands that we want to execute had to be written out in the program. We will now write a program which will allow the user to control the turtle during *during execution*.

Let us decide that the user must type the character 'f' to make the turtle go forward by 100 pixels, the character 'r' to make the turtle turn right by 90 degrees, and the character 'l' to make the turtle turn left by 90 degrees. Our program must receive these characters that the user types, and then move the turtle accordingly. Here it is.

```
main_program{
```

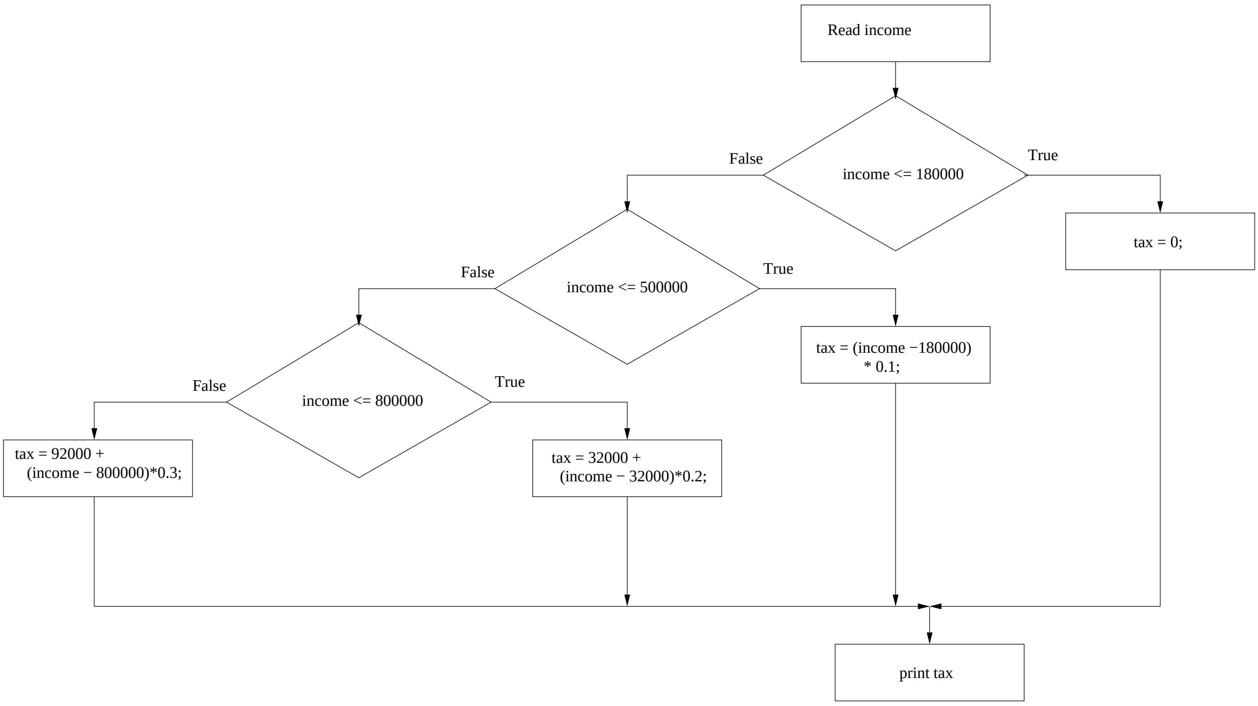


Figure 5.5: Flowchart for second income tax program

```
  char command;
  turtleSim();

  repeat(100){
    cin >> command;
    if (command == 'f') forward(100);
    else if (command == 'r') right(90);
    else if (command == 'l') left(90);
    else cout << "Not a proper command, " << command << endl;
  }
}
```

Remember that `char` data is really numerical, so it is perfectly acceptable to compare it using the operator `==`. This program will execute 100 user commands to move the turtle before stopping. Try it!

5.4.1 "Buttons" on the canvas

We can build "buttons" on the canvas using the `Rectangle` shapes of Section 4.3. We can control the turtle by clicking on the buttons. This gives yet another turtle controller.

```
main_program{
  initCanvas();
  const float bFx=150,bFy=100, bLx=400,bLy=100, bWidth=150,bHeight=50;
  Rectangle buttonF(bFx,bFy,bWidth,bHeight), buttonL(bLx,bLy,bWidth,bHeight);

  Text tF(bFx,bFy,"Forward"), tL(bLx,bLy,"Left Turn");
  Turtle t;

  repeat(100){
    int clickPos = getClick();
    int cx = clickPos/65536;
    int cy = clickPos % 65536;

    if(bFx-bWidth/2<= cx && cx<= bFx+bWidth/2 &&
       bFy-bHeight/2 <= cy && cy <= bFy+bHeight/2) t.forward(100);

    if(bLx-bWidth/2<= cx && cx<= bLx+bWidth/2 &&
       bLy-bHeight/2 <= cy && cy <= bLy+bHeight/2) t.left(10);
  }
}
```

The program begins by drawing the rectangles on the screen. Notice that we have not given the coordinate information of the buttons by writing numbers directly, but first created the names `bFx,bFy` and so on having specific values and then used these names in the button creation. Using such names is convenient: if you want to adjust the layout of buttons later, you just need to change the value of some name. Without names, you would have needed to make changes in every place the number appeared. In the present case, if you want to change the width of the rectangles, you just need to assign a different value to `bWidth`, instead of worrying in which all places the width value needs to be changed.

Next, text is put in the rectangles. Then we go into a loop. Inside, we wait for the user to click. We check whether the click is inside either of the two rectangles. This is done in the two `if` statements in the loop. Each check has two parts: we must check if the x coordinate of the click is between the left edge of the rectangle and the right edge, i.e. the left edge coordinate must be smaller or equal, and the right edge coordinate must be larger or equal. And correspondingly we must check for the y coordinate as well.

This program will only allow 100 clicks; we see later how to make the loop indefinitely or stop if some condition is met.

5.5 The `switch` statement

In the turtle control program, there was a single variable, `command`, depending upon which we took different actions. A similar situation arises in many programs. So C++ provides the `switch` statement so that we can express our code succinctly. The general form of the `switch` statement is:

```
switch (expression){
  case constant1:
     group(1) of statements usually ending with ‘‘break;’’
  case constant2:
     group(2) of statements usually ending with ‘‘break;’’
  ...
  default:
     default-group of statements
}
```

The portion consisting of `default:` and the group of statements following that is optional. Each `constati` in above is required to be an integer constant.

The statement executes in the following manner. First the `expression` is evaluated. If the value is identical to `constanti` for some i, then we start executing `group(i)` statements. We execute `group(i)` statements, then `group(i+1)` statements and so on, including `default-group` statements, unless we encounter a `break;` statement. If we encounter a `break` then the execution of the `switch` is complete, i.e. we do not execute the statements following the `break` but directly go to the statement in the program following the `switch` statement. If the value of `expression` is different from any of the `constant` values mentioned, then the `default-group` of statements is executed.

If a certain `group(i)` does not end in a `break`, then the execution is said to "fall-through" to the next group. Fall-throughs are considered to be rare.

Using a `switch` our turtle control program can be written as follows.

```
main_program{
  char command;
  turtleSim();

  repeat(100){
    cin >> command;
    switch(command){
      case 'f': forward(100);
                break;
      case 'r': right(90);
                break;
      case 'l': left(90);
                break;
      default: cout << "Not a proper command, " << command << endl;
    }
  }
}
```

As you can see the new program is nicer to read.

Here is an example which has fall-throughs. Suppose we want to print the number of days in the nth month of the year, taking n as the input. Here is the program.

```
main_program{
  int month;
  cin >> month;
  switch(month){
    case 1:  // January
    case 3:  // March
    case 5:  // May
    case 7:  // July
    case 8:  // August
    case 10: // October
    case 12: // December
             cout << ‘‘This month has 31 days.\n’’;
             break;
    case 2:  // February
             cout << ‘‘This month has 28 or 29 days.\n’’;
             break;
    case 4:  // April
    case 6:  // June
    case 9:  // September
    case 11: // November
             cout << ‘‘This month has 30 days.\n’’;
             break;
    default: cout << ‘‘Invalid input.\n’’;
  }
}
```

Suppose the input is 5. Then the execution will start after the point labelled `case 5:`. It will fall through the cases 5,7,8,10 to case 12. In this the number of days will be printed to be 31, and then a `break` is encountered. This will complete the execution of the `select`.

The `switch` statement is considered somewhat error-prone because you may forget to write `break;`. So be careful.

5.6 Conditional Expressions

C++ has a notion of a *conditional expression*, having the following form.

```
condition ? consequent-expression : alternate-expression
```

The evaluation of this proceeds as follows. First the `condition` is evaluated. If it is `true`, then the `consequent-expression` expression is evaluated, and that is the value of the overall expression. The `alternate-expression` is ignored. If on the other hand the `condition` evaluates to false, then the `consequent-expression` is ignored, the `alternate-expression` is evaluated and the resulting value is the value of the overall expression.

Here is are some examples.

```
int marks; cin >> marks;
int actualmarks = (marks > 100) ? 100 : marks;
int nonsense = (marks % 2 == 0) ? marks*2 : marks*3;
```

In this if marks read in were more than 100, then actualmarks would be capped to 100, else actualmarks would be set equal to marks. The variable nonsense would be set to the value marks*2 if marks % 2 == 0, i.e. if marks is even, and to marks*3 if marks is odd.

Conditional expressions can be nested, i.e. the consequent or alternate expressions can themselves conditional expressions. This allows us to write a very compact but unreadable tax calculation program.

```
main_program{
  float income; cin >> income;

  cout << (
           income <= 180000 ? 0 :
           income <= 500000 ? (income - 180000) * 0.1 :
           income <= 800000 ? 32000 + (income - 500000) * 0.2 :
           92000 + (income - 800000) * 0.3
          )
       << endl;
}
```

We merely read in the income, and then calculate the tax as an expression and directly print it out without storing it into a variable. In the above the big conditional expression is parenthesized. This is because the operator << has higher precedence than the operator <=.

The above program is very compact, but not recommended. Most programmers would consider it unreadable. However, the conditional expression without nesting is considered to be a useful construct.

5.7 Logical Data

An important part of the if statement are the conditions. We have already seen that a condition is either true or false, i.e. we can associate the value true or the value false with each condition. We have also seen that conditions can be combined in different ways. The resulting combination will also be true or false. We have also seen that there may be several equivalent ways of writing the same condition (as we saw for the second if statement of our first program). In this sense, conditions are similar to numerical expressions, numerical expressions have a value, numerical expressions can be combined to build bigger numerical expressions, we can have numerical expressions that are equivalent. In that case, why not treat conditions, as just another kind of data? This turns out to be a very good idea, and an algebra for manipulating conditions, or what we will hereafter refer to as logical expressions was developed by George Boole in 1940. C++ supports the manipulation and storage of logical data, and in honour of Boole the data-type for storing logical data is named bool. You have already seen this data type in Chapter 3, now we will do more interesting things with it.

First we note that we can assign values of logical expressions to bool variables. Consider the following code.

```
float income; cin >> income;
```

```
bool lowIncome, midIncome, highIncome;
lowIncome = (income <= 180000);
midIncome = (income > 180000) && (income <= 800000);
highIncome = (income > 800000);
```

Suppose during execution the value 200000 is given for `income`. Then after the execution of the subsequent statements, the variables `lowIncome, midIncome, highIncome` would respectively have the values `false, true, false`.

As you can see, the right hand sides of the above assignment statements are conditions, and whatever the values these conditions have will been put in the corresponding left hand side variables.

As another example, let us define a `bool` variable that will be `true` if a character read from `cin` happens to be a lower case character. Note that this will happen if the ASCII value of the character is at least 'a' and at most 'z'. Thus the code for this could be

```
char in_ch;
bool lowerCase;
cin >> in_ch;
lowerCase = (in_ch >= 'a') && (in_ch <= 'z');
```

We will next consider a more complex program which determines whether a given number `num` is prime or composite. The ability to store logical values will be useful in this program. To understand that program we will need to reason about expressions containing logical data. So we first discuss this.

5.7.1 Reasoning about logical data

As we discussed earlier, the same condition can be expressed in many ways. It is important to understand which expressions are equivalent.

First, let us make a few simple observations. For any logical value `v`, we have that `v || false` has the same value as `v`. The easiest way to check this is to try out all possibilities: if `v` is true, then `true || false` is clearly `true`. If `v` is `false`, then `false || false` is clearly `false`. Thus `false` plays the same role with respect to `||` that 0 plays with respect to numerical addition. More formally, `false` is said to be the identity for `||`. Likewise `true && v` has the value `v` for any `v`. Or in other words, `true` is the identity for `&&`.

Another rule is the so called distributivity of `&&` over `||`. Thus, if `x,y,z` are boolean variables (or equivalently, conditions), then `(x && y) || z` is the same as `(x && z) || (y && z)`. In a similar manner, it turns out that `||` also distributes over `&&`.

Another important rule is that `x || !x` is always true, and hence we can replace such expressions with `true`. Similarly, `x && !x` can be replaced with `false`.

Finally, an important rule is DeMorgan's Law. This says that `!x && !y` is the same as `!(x || y)`. Similarly `!x || !y` is the same as `!(x && y)`.

Consider first a condition such as `income <= 180000`. Income being at most 180000 is the same as it not being bigger than 180000. Hence we can write this condition also as `!(income > 180000)`.

While it is fine to be able to intuitively understand that the conditions

```
(income >  180000) && (income <= 500000)
```

and

```
!((income <=  180000) || (income > 500000))
```

are the same, you should also be able to deduce this given the rules given in this section.

5.7.2 Determining whether a number is prime

Determining whether a number is prime is an important problem, for which very sophisticated, very fast algorithms are known. We will only consider the simplest (and hence substantially slower than the fastest known) algorithms in this book.

Here is the most obvious idea. We go by the definition. A number n is prime if it has no divisors other than itself and 1. So it should suffice to check whether any number i between 1 and itself (both exclusive) divides it. If we find such an i then we declare x to be composite; otherwise it is prime.

This requires us to generate all numbers between 2 and $x-1$ (both inclusive this time) so that we can check whether they divide x. This is really the sequence generation pattern (Section 3.4.1) which we saw, say in the spiral drawing program of Section 3.3. There we made `i` take 10 values starting at 1. Now we want `i` to take the $x-2$ values from 2 to $x-1$. So here is the code fragment we should use:

```
i=2;
repeat(x-2){
  /*
  Here i takes values from 2 to x-1.
  */
  i = i + 1;
}
```

In each iteration of the loop we can check whether `i` divides `x`. This is really the condition `(x % i) == 0`. We want to know whether any such condition succeeds. But this is nothing but a logical or, of the conditions that arise in each iteration. In other words, this itself is the accumulator pattern mentioned in Section 3.4.1. But we know how to implement that! We saw how to do it to calculate the sum in the average computing program of Section 3.3. We must maintain an accumulator variable which we set to the identity for the operator in question, and we update it in each step. Say we name our accumulator variable `found` (since it will indicate whether a factor is found). Then we initialize it to `false`, the identity for the OR operation. Then in each step of the loop we merely update found, exactly as we updated `sum` in the average computation program. So our code fragment becomes:

```
i=2;
found = false;
repeat(x-2){
  found = found || (x % i) == 0;
  i = i+1;
}
```

At the end of this found will indeed be true if any of the expressions `(x % i) == 0` was `true`, for any value of `i`. Thus following this code we simply print prime/composite depending upon whether found is `false`/`true`. And at the beginning we need to read in `x` etc. The complete program is as follows.

```
main_program{  //Decide if x is prime.
  int x; cin >> x;

  int i=2;
  bool found = false;
  repeat(x-2){
    found = found || (x % i) == 0;
    i = i+1;
  }

  if (found) cout << x << " is composite." << endl;
  else cout << x << " is prime." << endl;
}
```

This program will be improved in several ways later. Once we find a factor of `x`, i.e. if in some iteration `x % i == 0` becomes true, we will set `found` to `true`, and no matter what we do later, it cannot become false. So why even do the remaining iterations? This is indeed correct: if we are testing if 102 is prime, we will discover in the first iteration itself that 102 is divisible by 2, i.e. 2 is a factor and that 102 is composite. So we should prematurely stop the loop and not do the remaining iterations. In the next chapter we will see how this can be done.

Note by the way that effect of `found = found || (x % i) == 0;` can also be had by writing `if(x % i == 0) found = true;` This doesnt look like accumulation, but has the same effect.

5.8 Remarks

There is a potential pitfall associated with the use of the operators `=` and `==`. In mathematics, the operator `=` is used to denote comparison, and since most of us learn mathematics before programming, we are likely predisposed to use `=` to mean comparison even in C++, rather than `==`. This will lead to errors. The situation is more serious than what you might think at first glance. If you write code such as

```
if(p = 25) q = 37;
```

when you mean `if(p == 25) q = 37;` the compiler will not regard it as an error. This is because assignment is also an expression, and in this case, `p = 25`, the value is 25. The compiler will, on its own, try to convert this value to a boolean value. For this the rule of conversion is a bit non-intuitive: any non-zero value becomes `true` and only 0 becomes `false`. Thus in the execution of the above statement, the assignment `q=37` will always happen.

Many compilers can be asked to warn if they encounter such statements which most likely are silly mistakes made by the programmer. Indeed the GNU C++ compiler will give a warning if it sees such statements in your program, provided you invoke it using the option `-Wparentheses`. And in fact, `s++` which you use with `simplecpp` indeed calls the GNU C++ compiler with this option, so you will get these warnings already if you compile with `s++`. If you really intended the statement to mean the assignment expression (and did not mistakenly write `=` instead of `==`), then you can merely put the expression inside a pair of parentheses and write `if((p = 25)) q = 37;`. This effectively declares your firm intent that you mean `p = 25` to be an assignment expression. Thus in this case no warning will be issued even if you use the option `-Wparentheses`.

Another pitfall concerns nesting of `if` statements, say if the `consequent` of an `if` is itself another `if` statement.

```
if(a > 0) if(b > 0) c = 5; else c = 6;
```

This is treated by the compiler to mean

```
if(a > 0) {if(b > 0) c = 5; else c = 6;}
```

In other words, the `else` joins with the innermost `if`, and the outer `if` is left without an `else` clause. Keeping track of such rules is rather cumbersome, so it is best if you insert the braces yourself. Of course if you meant to associate the else with the outer `if` you could have written

```
if(a > 0) {if(b > 0) c = 5;} else c = 6;
```

If you omit the braces, then the compiler again will warn you if you have used the `-Wparentheses` option. Note that the compiler will have compiled your program as per the rules of C++, even when it issues a warning. However, you should treat compiler warnings as suggestions to improve the readability of your code. Indeed, if you use parentheses or braces as suggested above, you make your code more readable to other programmers as well.

5.9 Exercises

1. Modify the turtle program so that the user can specify how many pixels the turtle should move, and also by what angle to turn. Thus if the user types “f100 r90 f100 r90 f100 r90 f100” it should draw a square. You may put spaces or newlines in the sequence for readability.

2. Write a program which takes as input a number denoting the `year`, and says whether the year is a leap year or not a leap year.

3. Write a program that takes as input a number `y` denoting the year and a number `d`, and prints the date which is the `d`th day of the year `y`. Suppose `y` is given as 2011 and `d` as 62, then your program should print “3/3/2011”.

4. Write a program that reads 3 numbers and prints them in non-decreasing order.

5. Suppose we wish to write a program that plays cards. The first step in such a program would be to represent cards using numbers. In a standard deck, there are 52 cards, 13 of each suite. There are 4 suites: spades, hearts, diamonds, and clubs. The 13 cards of each suit have the denomination 2,3,4,5,6,7,8,9,10,J,Q,K,A, where the last 4 respectively are short for jack, queen, king and ace. It is natural to assign the numbers 3,2,1,0 to the suites respectively. The denominations 2 – 10 are assigned numbers same as the denomination, whereas the jack, queen, king, and ace are respectively assigned the numbers 11, 12, 13, and 1 respectively. The number assigned to a card of suite s and denomination d is then $13s + d$. Thus the club ace has the smallest denomination, 1, and the spade king the highest, 52. Write a program which takes a number and prints out what card it is. So given 20, your program should print "7 of diamonds", or given 51, it should print "queen of spades".

6. Write a program that takes a character as input and prints 1 if it is a vowel and 0 otherwise.

7. Can you write the program to determine if a number is prime without using a `bool` variable? Hint: count how many factors the number has.

8. A number is said to be perfect if it is equal to the sum of all numbers which are its factors (excluding itself). So for example, 6 is perfect, because it is the sum of its factors 1, 2, 3. Write a program which determines if a number is perfect. It should also print its factors.

9. Write a program which prints all the prime numbers smaller than n, where n is to be read from the keyboard.

10. Suppose you want to add two 64 bit (unsigned) numbers. Here is how you could do it. First, you need to decide how to store the numbers. One possibility to store the least significant 32 bits in an `unsigned int` variable, and the more significant 32 bits in another `unsigned int` variable. This is equivalent to representing the number using the radix 2^{32}, and each variable storing one digit (which has 64 bits) in each long variable. Suppose now that we have stored the most and least significant bits of the first number in variables `ams32` and `als32`, and similarly the second number in variables `bms32` and `bls64`. Say we want to get the sum in variables `cms64` and `cls64`. We can obtain the least significant 32 bits merely by writing `csls32 = als32 + bls32;` – however we also need the carry out of the least significant 32 bits. What condition will you check to determine what the carry will be? Note that if you do arithmetic on 32 bit integers, you basically get the result modulo 2^{32}. Choose a suitable radix.

11. How would you multiply two 64-bit numbers? Hint: If `a,b` are `unsigned short` and `c` is `unsigned int`, then `c=a*b` will indeed get the 32 bit product of `a,b` into `c`.

12. Make an animation of a ball bouncing inside a rectangular box. Assume that the box is placed on the ground, and the ball moves horizontally inside, without friction. Further assume for simplicity that the ball has an elastic collision with the walls of the box, i.e. the velocity of the ball parallel to the wall does not change, but the

velocity perpendicular to the wall gets negated. Put the pen of the ball down so that it traces its path as it moves. You can either read the ball position and velocity from the keyboard, or you can take it from clicks on the canvas. Move the ball slowly along its path so that the animation looks nice.

13. Modify the animation assuming that the box has mass equal to the ball, and is free to move in the x direction, say it is mounted on frictionless rails parallel to the x direction. Note that now in each collision the velocity of the box will also change. Show the animation of this system. You may want to start off the system such that the total momentum in the x direction is zero, thereby ensuring that the box doesnt move out of the screen.

14. * In the hardest version of the ball in a box problem the box is sitting on a frictionless surface, and is free to turn. Now after a collision, the box will in general start rotating as well as translating. Assume for simplicity that the mass of the box is uniformly distributed along its 4 edges, i.e. the base is massless.

15. A *digital* circuit takes as input electrical signals representing binary values and processes them to generate some required values, again represented as electrical signals. As discussed in Section 2.2, a common convention is to have a high voltage value (e.g. 0.7 volts) represent 1, and a low value (e.g. 0 volts) represent a 0. A digital circuit is made out of components customarily called *gates.* An AND gate has two inputs and one output. The circuit in an AND gate is such that the output is 1 (i.e voltage at least 0.7 volts) if both inputs are 1. If any input is a 0 (i.e. voltage 0 volts or less), then the output is a 0. Likewise, an OR gate also has 2 inputs and a single output. The output is a 0 if both inputs are 0, and it is one if even one of the inputs is a 1. An XOR gate has 2 inputs and one output, and the output is 0 iff the inputs are both the same value. Finally, a NOT gate has one input and one output, and the output is 1 if the input is 0, and 0 if the input is 1.

    Figure 5.6 shows the symbols for the NOT gate, the AND gate and the OR gate at the top, left to right, and a circuit built using these gates at the bottom.

    The inputs to a digital circuit are drawn on the left side, and the outputs on the right. The gates are placed in between. A gate input must either be connected a circuit input or to the output of some gate to its left. The gate input takes the same value as the circuit input or the output to which it is connected. In the figure, a, b, c are the circuit inputs. Some gate outputs are designated as circuit outputs, e.g. outputs p, q. Note that if two wires in the circuit cross, then they are not deemed to be connected unless a solid dot is present at the intersection.

    In this exercise, you are to write a program which takes as inputs the values of the circuit inputs a, b, and prints out the values of the outputs c, d.

16. Develop a mini drawing program as follows. Your program should have buttons called “Line” and “Circle” which a user can click to draw a line or a circle. After a user clicks on “Line”, you should take the next two clicks to mean the endpoints of the line, and so after that a line should be drawn connecting those points. For now, you will have to `imprint` that line on the canvas. Similarly, after clicking “Circle” the next point

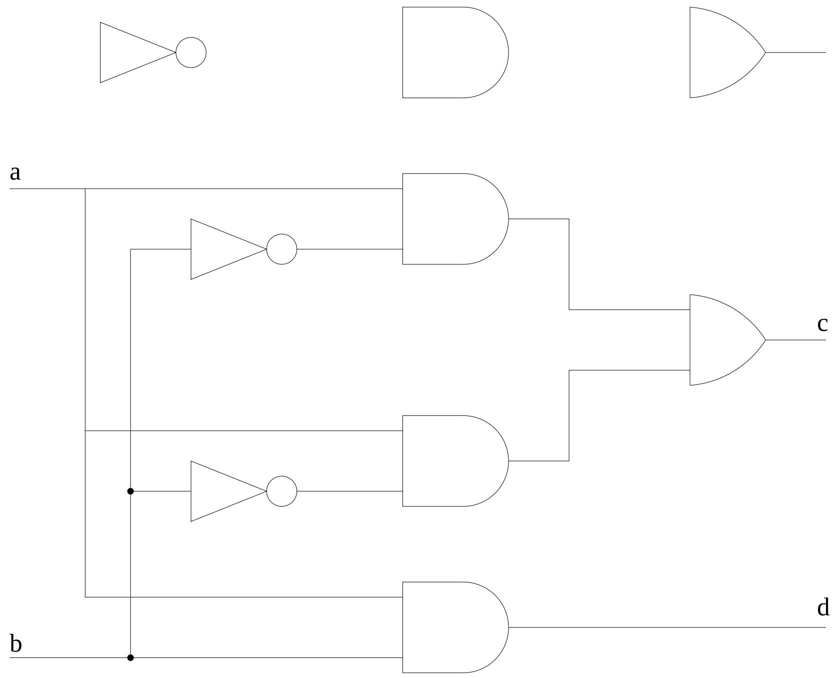


Figure 5.6: Circuit components and a circuit

should be taken as the center, and the next point as a point on the circumference. You can also have buttons for colours, which can be used to select the colour before clicking on “Line” or “Circle”.

Chapter 6

Loops

Consider the following *mark averaging* problem:

> From the keyboard, read in a sequence of numbers, each denoting the marks obtained by students in a class. The marks are known to be in the range 0 to 100. The number of students is not told explicitly. If any negative number is entered, it is not to be considered the marks of any student, but merely a signal that all the marks have been entered. Upon reading a negative number, the program should print the average mark obtained by the students and stop.

Using the statements you have learned so far, there is no nice way in which the above program can be written. It might seem that the program requires us to do something repeatedly, but the number of repetitions equals the number of students, and we dont know that before starting on the repetitions. So we cannot use the `repeat` statement, in which the number of times to repeat must be specified before the execution of the statement starts.

In this chapter we will learn the `while` loop statement which will allow us to write the program described above. We will also learn the `for` loop statement, which is a generalized version of the `while` statement. All the programs you have written earlier using the `repeat` statement can be written using `while` and `for` instead, and often more clearly. The `repeat` statement is not really a part of C++, but something we added through the package `simplecpp` because we didnt want to confuse you with `while` and `for` in the very first chapter. But having understood these more complex statements you will find no real need for the `repeat` statement. So we will discontinue its use from the next chapter.

6.1 The `while` statement

The most common form of the `while` statement is as follows.

```
while (condition) body
```

where `condition` is a boolean expression, and `body` is a statement, including a block statement. The `while` statement executes as follows.

1. The `condition` is evaluated.

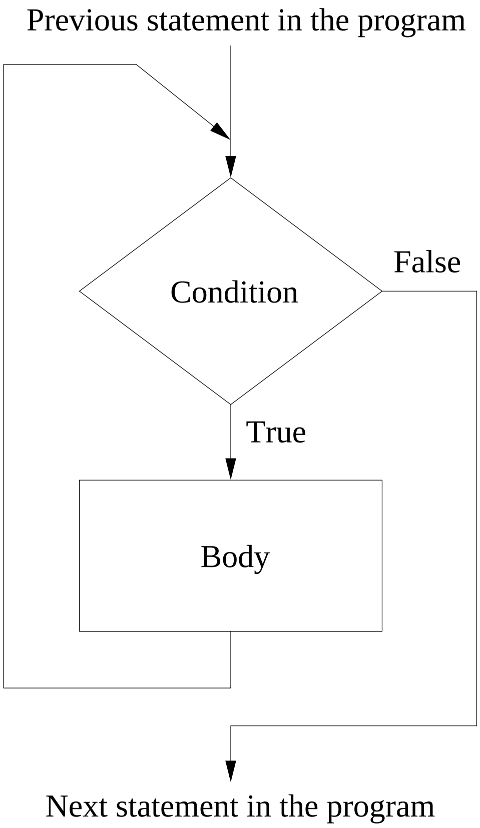

Previous statement in the program
Condition
False
True
Body
Next statement in the program

Figure 6.1: While statement execution

2. If the condition is `false`, sometimes described as "if the condition fails", then the execution of the statement is complete without doing anything more. Then we move on to execute the statement following the `while` statement in the program.

3. If the `condition` is `true`, then the `body` is executed.

4. Then we start again from step 1 above.

This is shown as a flowchart in Figure 6.1.

Each execution of the `body` is called an iteration, just as it was for the `repeat`. Note that typically each iteration might change the values of some of the variables so that eventually `condition` will become `false`. When this happens, it will be detected in the subsequent execution of step 1, and then step 2 will cause the execution of the statement to terminate.

As you can perhaps already see, this statement is useful for our marks averaging problem. But before we look at that let us take a few simple examples.

First we note that using a `while`, it is possible to do anything that is possible using a `repeat`. To illustrate this, here is a program to print out a table of the cubes of numbers from 1 to 100. Clearly you can also write this using `repeat`.

```
main_program{
  int i=1;
  while(i <= 100){
    cout << ‘‘The cube of ‘‘ << i << ‘‘ is ‘‘ << i*i*i << endl;
    i = i + 1;
  }
  cout << ‘‘Done!’’ << endl;
}
```

The execution will start by setting i to 1. Then we check whether i is smaller than or equal to 100. Since it is, we enter the body. The first statement in the body causes us to print "The cube of 1 is 1", because i has value 1. Then we increment i. After that we go back to the top of the statement, and check the condition again. Again we discover that the current value of i, 2, is smaller than or equal to 100. So we print again, this time with i=2, so what gets printed is "The cube of 2 is 8". We again execute the statement i = i + 1;, causing i to become 3. We then go back and repeat everything from the condition check. In this way it continues, until i is no longer smaller than or equal to 100. In other words, we execute iterations of the loop until (and including) i becomes 100. When i becomes 101, the condition i >= 100 fails, and so we go to the statement following the loop. Thus we print "Done!" and stop. But before this we have executed the loop body for all values of i from 1 to 100. Thus we will have printed the cube of all the numbers from 1 to 100.

6.1.1 Counting the number of digits

We consider one more problem: read in a non-negative integer from the keyboard and print the number of digits in it. The number of digits in a number n is simply the smallest positive integer d such that $10^d > n$. So our program could merely start at $d = 1$, and try out successive values of d until we get to a d such that $10^d > n$.

Thus we have to generate the sequence $10, 10^2, 10^3, \ldots$; but this is just the sequence generation idiom. We should stop generating the sequence as soon as we generate a sequence element, say 10^d which is larger than n. In other words, we should not stop while $10^d \leq n$. This is what the following code does.

```
main_program{
  int n;
  cout << "Type a number: ";
  cin >> n;

  int d = 1, ten_power_d=10;
  while(n >= ten_power_d){
    // invariant: ten_power_d is 10 raised to d on entry to the loop.
    d++;
    ten_power_d *= 10;
  }

  cout << "The number has " << d << " digits." << endl;
}
```

Let us see what happens when we run the program. Say in response to the request to type in a number, we entered 27. Then we would set d to 1 and ten_power_d to 10. Then we would come to the while loop. We would find that n, which equals 27 is indeed bigger than or equal to ten_power_d) which equals 10. So we enter the loop. Inside the loop, we add 1 to d so that it becomes 2, and we multiply ten_power_d by 10, so it becomes 100. We then go back to the beginning of the loop and check the condition. This time we would find that n whose value is 27 is smaller than ten_power_d whose values is 100. So we do not enter the loop but instead go to the statement following the loop. Thus we would print the current value of d, which is 2, as the number of digits. This is the correct answer: the number of digits in 27 is indeed 2.

Clearly, the invariant for the program is: on entry to the loop, the value of ten_power_d is 10^d where d is the value of d. You can use this to argue that the program is correct in general.

6.1.2 Determining if a number is prime

In Section 5.7.2 we developed a program to determine if a number is prime. We remarked there that the program can be made more efficient by noting that once we find a factor for the given number, we can stop checking for additional factors and immediately report that the number is composite. We can implement this idea using the while statement as follows.

```
main_program{  //Decide if x is prime.
  int x; cin >> x;

  int i=2;
  bool found = false;
  while((i < x) && !found){
    found = found || (x % i) == 0;
    i = i+1;
  }

  if (found) cout << x << " is composite." << endl;
  else cout << x << " is prime." << endl;
}
```

As you can see, the new program is only slightly different, insteas of repeat(x-2) we have while((i < x) && !found). The first part of the new condition, i < x says that we must repeat till i becomes bigger than x. If we just had this, i.e. we had written while(i < x), then the code would have behaved identically to the code in Section 5.7.2. This is because i becomes x only after x-2 iterations, which is precisely the repeat count in Section 5.7.2. However, in the new code, we also have the additional clause: !found. Thus, in order for the iteration to continue, we also need !found to be true, or found to be false. In other words, we will stop immediately after the iteration in which found becomes true, if it does. But found becomes true only when we find a factor of x, at which we can conclude that x is composite and dont need to check any further.

We can further improve the program by observing that if x is composite then it must have at least one factor no larger than $\sqrt{x}$. Thus we can terminate the while loop above as soon as i becomes bigger than the square root of x, which is the same thing as saying that i*i becomes bigger than x. We can get implement this by stating the condition for the while to be ((i*i <= x) && !found) instead of just ((i < x) && !found). If you do this, in each iteration of the program you will have to compute i*i, however, this increase in the computation time will be outweighed because of the reduction in the number of iterations to about $\sqrt{x}$.

6.2 Developing loop based programs

The programs given in the previous section are quite intuitive, howeve, in general, developing loop based programs is trickier.

When developing loop based programs, there are 3 main issues to worry about. First, we must identify the pattern of repetitive actions. These actions will go into the body of the loop. As we will see in Section 6.2.2, this may require some adjustment. Second, we must identify the condition under which the repetition should be stopped. The *negation* of this condition will become the condition in the while statement. Finally, each iteration of the loop will access the same variables; we have to organize our code so that we can *reuse* the variables going from one iteration to the next, i.e. the body of the loop must prepare the variables for being used again in the next iteration as needed. This is sometimes tricky, as will be seen in Section 6.2.1.

In this section we start by developing the program for the problem of Virahanka numbers. Following that we discuss the mark averaging problem.

6.2.1 Virahanka numbers

Consider a sequence V_i, where i is any natural number, defined as follows.

$$V_i = \begin{cases} 1 & \text{if } i = 1 \\ 2 & \text{if } i = 2 \\ V_{i-1} + V_{i-2} & \text{if } i > 2 \end{cases}$$

This definition of V_i is said to be a recurrence; in general in a recurrence the ith term of a sequence is defined using the preceding terms. The cases directly defined, V_1, V_2 above, are said to be the *base* cases.

This sequence is more popularly known as the Fibonacci sequence, but Virahanka wrote about it earlier than Fibonacci. We will discuss the context in which Virahanka discovered this sequence later in Section 10.3. For now we will consider how to compute its elements. How do we compute the nth Virahanka number V_n? V_1, V_2 are known immediately, but for larger values of i, we need to do some work.

Clearly, we can compute V_3 using the recurrence and base cases: $V_3 = V_2 + V_1 = 2+1 = 3$. After that we can compute $V_4 = V_3 + V_2 = 3 + 2 = 5$. After that we can compute V_5, and this process can go on. Clearly, there is something repetitive going on here. So presumably a loop will be useful to program it. Presumably, we can compute one Virahanka number in each iteration, and there need to be $n - 2$ iterations, because V_1, V_2 were computed before entering the loop.

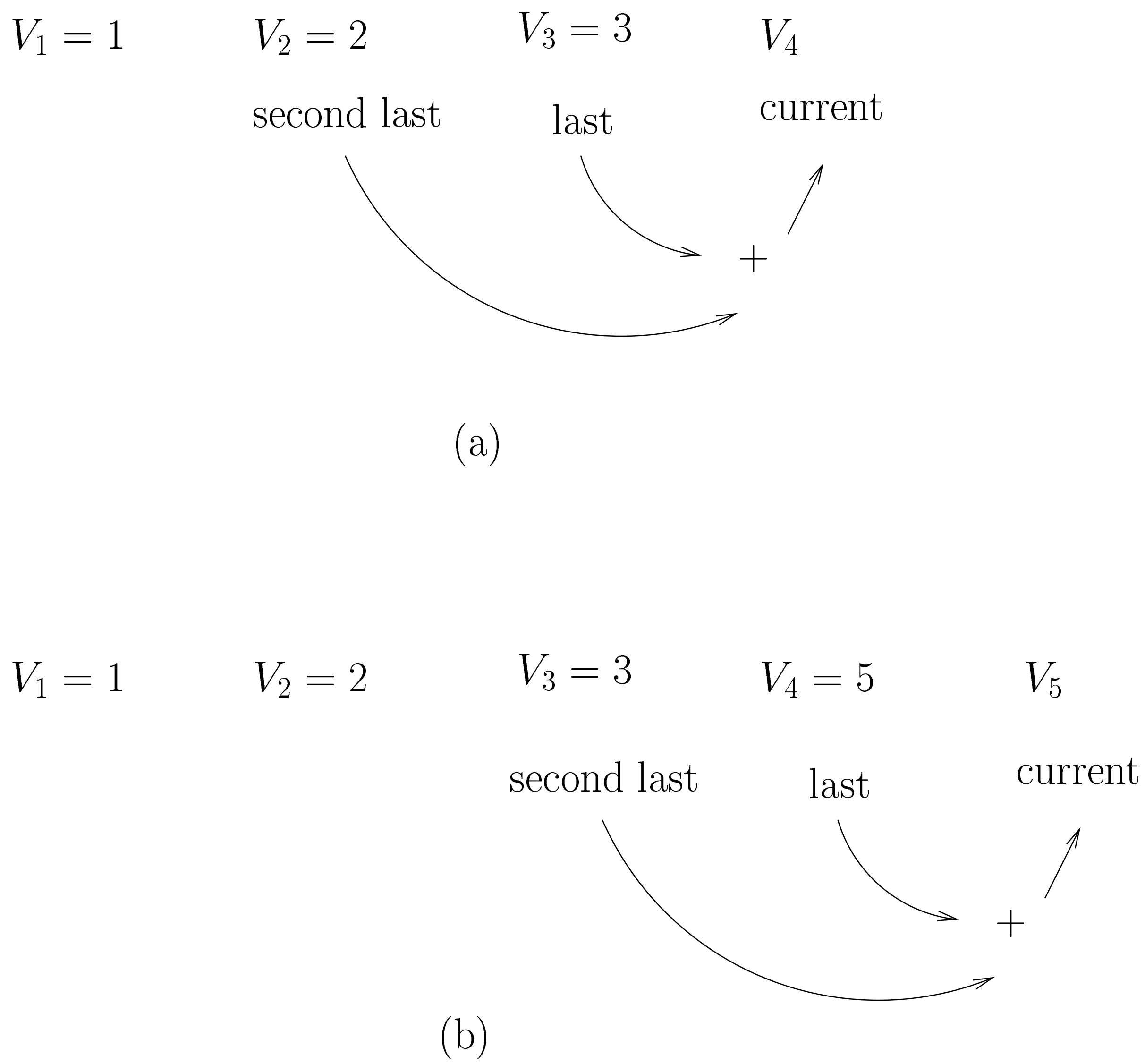


Figure 6.2: Computing Virahanka Numbers

To calculate a number in the current iteration, we need to add the numbers calculated in the last iteration, and the second last iteration, as shown in Figure 6.2. The key point, however, is that what is "last" in one iteration, e.g. Figure 6.2(a) is called "second last" in the next iteration, Figure 6.2(b). Similarly, the "current" of Figure 6.2(a) becomes "last" of Figure 6.2(b). We have to keep this in mind while writing the code.

In the code, we will have variables `secondlast`, `last`, which we will assume already have values, and which we will add to produce a value which will be placed in a variable `current`. To go to the next iteration, as shown in Figure 6.2 our notion of what is last and second last must change. So accordingly we change their values. This is the body of the loop, which must be executed $n-2$ times. So here is the possible program.

```
main_program{                     // Program to compute Virahanka number V_n
  int n;
  cin >> n;

  int v1=1,v2=2;                  // v1 = V_1,  v2 = V_2
  if(n == 1) cout << v1 << endl;
  else if(n == 2) cout << v2 << endl;
  else {
    int secondlast=v1, last=v2, current;
    repeat(n-2){
      current = secondlast + last;

      secondlast = last;  // prepare for next iteration
      last = current;
    }

    cout << current << endl;
  }
}
```

The important point to note in this example is the preparation of the variables `secondlast` and `last` for the next iteration. This also stresses the important point that we should consider each variable as performing a certain function, e.g. holding the last computed Virahanka number, and the value of the variable must be changed to reflect its function.

We can of course rewrite this as a `while` loop.

```
main_program{                     // Program to compute Virahanka number V_n
  int n;
  cin >> n;

  int v1=1,v2=2;                  // v1 = V_1,  v2 = V_2
  if(n == 1) cout << v1 << endl;
  else if(n == 2) cout << v2 << endl;
  else {
    int secondlast=v1, last=v2, current;
    int i=0;
    while(i < n-2){
      current = secondlast + last;

      secondlast = last;  // prepare for next iteration
      last = current;
      i++;
    }

    cout << current << endl;
  }
}
```

6.2.2 Mark averaging

This problem, like many problems you will see later, is what we might call a *data streaming* problem. By that we mean that the computer receives a stream (sequence) of values, and we are expected to produce something when the stream terminates. Occasionally, we may be expected to print out a stream of values as well, but in the current problem, we have to only print out their average. A general strategy for tackling such problems is to ask yourself: what information do I need to remember at a point in execution when some n values of the stream have been read? The answer to this often suggests what variables are needed, and how they should be updated.

For the mark averaging problem, we know what we want at the end: we want to print out the average. To calculate the average we need to know the sum of all the values that we read, and a count of how many values we read. So at an intermediate point in the program, when some n values have been read, we should keep track of n as well as their sum. We dont need to remember the individual values that we have read so far! So it would seem that we should keep a variable `sum` in which we maintain the sum of the values that we have read till any point in time. We should also maintain a variable `count` which should contain the number of values we read. Both variables should start off 0. We will have a repeated portion in which we read a value, and for this we will have a variable called `nextmark`. Using these it would seem that we need to do the following steps repeatedly.

1. Read a value into `nextmark`.

2. If `nextmark` is negative, then we have finished reading, and so we go on to calculating and printing the average.

3. If `nextmark` is non-negative, then we add `nextmark` to `sum`, and also increment `count`.

4. We repeat the whole process from step 1.

In this we have not written down the process of calculating the average etc. But that is simply dividing `sum` by `count`. Figure 6.3(a) shows this as a flowchart.

Can we express this flowchart using the `while` statement? For this, you would need to match the pattern of the flowcharts of Figure 6.1 and Figure 6.3(a). It seems natural to

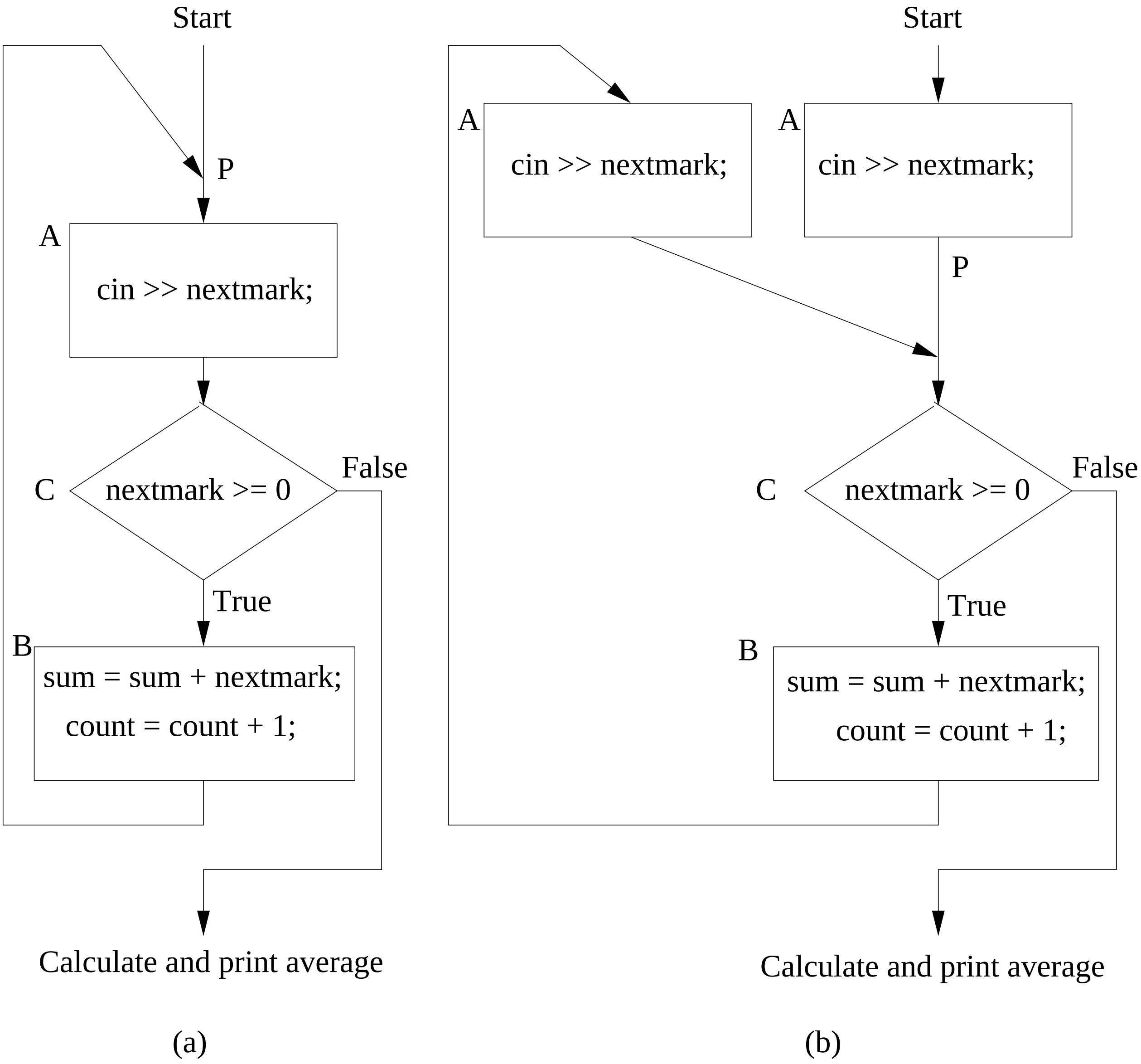


Figure 6.3: Flowcharts for averaging

match the condition in the former with the test nextmark >= 0 in the latter. But there is an important difference in the structure of the two flowcharts. In Figure 6.1 the condition test is the first statement of each iteration, while in Figure 6.3(a), the first statement is reading the data, and only the second statement is the condition check.

The crucial question then is: can we somehow modify the flowchart of Figure 6.3(a) so that the execution remains the same, but the new flowchart matches the pattern of Figure 6.1? Suppose we decide to move the box labelled A upwards above the point P where two branches merge. We do not want to change what happens on each branch that enters P, so then it simply means that we must place a copy of A on both branches coming into P. This gives us the flowchart of Figure 6.3(b). As you can see, the two flowcharts are equivalent in that they will cause the same statements to be executed no matter what input is supplied from the keyboard.

Note now that box B and the left copy of A in Figure 6.3(b) are executed successively, so we can even merge them into a single box containing 3 statements. This new box can become the body of a while statement, and box C the condition. Thus we can write our code as follows.

```
main_program{
  float nextmark, sum=0;
  int count=0;

  cin >> nextmark;         // right copy of box A

  while(nextmark >= 0){    // box C
    sum = sum + nextmark;  // Box B
    count = count + 1;     // Box B

    cin >> nextmark;       // left copy of box A
  }

  cout << ``The average is: `` << sum/count << endl;
}
```

The above program assumes that there will be at least one true mark, so that count will not be zero at the end.

Note the general idea carefully: the natural way of expressing our program could involve a test in the middle of the code we wish to repeat. In such cases, we can get the test to be at the top by moving around some code and also making a copy of it. Soon you will start doing this automatically.

6.3 The break statement

C++ allows a break; statement to be used inside the body of a while (both forms). The break statement causes the execution of the containing while statement to terminate immediately. If this happens, execution is said to have *broken* out of the loop. Here is a different way of writing our mark averaging program using the break statement:

```
float nextmark, sum=0;
int count=0;

while(true){
  cin >> nextmark;
  if(nextmark < 0) break;
  sum = sum + nextmark;
  count = count + 1;
}
cout << total/nmarks;
```

The first point to note here is that `condition` is given as `true`. This means that the statement will potentially never terminate! However, the statement does terminate because of the `break` statement in the `body`. After the `nextmark` is read, we check if it is negative – if so the statement terminates immediately, and we exit the loop. If the `nextmark` is non-negative, we add `nextmark` to `sum` and so on. The result of this execution will be the same as before. Note that this is similar to the flowchart of Figure 6.3(a).

Is the new program better than the old one? It is better in one sense: the statement `cin >> nextmark;` is written only once. In general it is a good idea to not duplicate code. First, this keeps the program small, but more importantly it prevents possible errors that might arise later. For example, suppose you later decide that just as you read `mark` you also want to print what was read. If the reading code is in several places, then you might forget to make the change in all the places. You may also think that the basic repetitive unit of work in the problem is read-process, rather than process-read, as it appears in the old code. So in this sense the new code is more natural.

The old code was better in that the `condition` for terminating the loop was given up front, at the top. In the new code, the reader needs to search a little to see why the loop will not execute *ad infinitum*. This could be cumbersome if the loop body was large. So we cannot unequivocally say that the new code is better.

6.4 The `continue` statement

What if someone typed in a number larger than 100 for `nextmark`? Since we are assuming that marks are at most 100, we could perhaps ignore the numbers above 100 as being erroneous. This is conveniently expressed using the `continue` statement.

When a `continue` statement is encountered during execution, the remaining part of the loop body is ignored. The control goes to the top of the loop, and checks the `condition` and begins the next iteration if check comes out `true`, and so on.

The main loop in the program can be written as follows using the `continue` statement.

```
while(true){
  cin >> nextmark;
  if(nextmark > 100){
    cout << "Larger than 100, ignoring." << endl;
    continue;
  }
  if(nextmark < 0) break;
  sum = sum + nextmark;
  count = count + 1;
}
```

If `nextmark` is bigger than 100, then the message is first printed, and then the rest of the loop body is skipped. The next iteration is begun, starting with the condition check, which in this case is always `true`.

6.5 The `do while` statement

The while statement has a variation in which the `condition` is tested at the end of the iteration rather than at the beginning. It is written slightly differently. The form is:

```
do body while (condition);
```

This is executed as follows.

1. The `body` is executed.
2. The `condition` is evaluated. If it evaluates to `true`, then we begin again from step 1. If the body evaluates to `false` then the execution of the statement ends.

In other words, in the `do-while` form, the body is executed at least once. You will observe that the `do-while` form above is equivalent to the following code using only the `while`:

```
body
while (condition) body
```

So you may wonder: why do we have this extra form as well? As you can see, the new form is more compact if you dont want the condition checked for the first iteration. Here is a typical example.

```
main_program{
  float x;
  char response;

  do{
    cout << ‘‘Type the number whose square root you want: ‘‘;
    cin >> x;
    cout << ‘‘The square root is: ‘‘ << sqrt(x) << endl;

    cout << ‘‘Type y to repeat: ‘‘;
    cin >> response;
    }
  while(response == ’y’);
}
```

This will keep printing square roots as long as you want.

6.6 The `for` statement

Suppose you want to print a table of cubes of the integers from 1 to 100. You would solve this problem using the following piece of code.

```
int i = 1;
repeat(100){
  cout << i << ‘‘ ‘‘ << i*i*i << endl;
  i = i + 1;
}
```

The variable `i` plays a central role in this code. All iterations of the `repeat` are identical, except for the value of `i`. Further, `i` changes from one iteration of the loop to another in a very uniform manner, in the above case it is incremented by 1 at the end of each iteration. This general code pattern: that there is a certain variable which takes a different value in each iteration and the value determines how the iteration will execute, is very common. Because of this, the designers of C++ (and other programming languages) have provided a mechanism for expressing this pattern very compactly. This mechanism is the `for` statement. Using the `for` statement, we can express the above code as follows.

```
for(int i=1; i <= 100; i = i + 1)
  cout << i << ‘ ‘ << i*i*i << endl;
```

This code is equivalent to the `repeat` loop above. Exactly why this is the case will become apparent when we understand the `for` statement in its general form:

```
for(initialization ; condition ; update) body
```

In this, `initialization` and `increment` are required to be expressions, typically assignment expressions. As you might remember, an assignment expression is simply assignments to a variable without including the semicolon, e.g. `i = i + 1`. Further we may include the definition along with the assignment e.g. `int i = 0`. As you might expect `condition` must be a boolean expression. The last part, `body` may be any C++ statement, including a block statement. In our example above, the `body` consisted of the statement `cout << i << ‘‘ ‘‘ << i*i*i << endl;`.

The execution of a `for` statement starts with the execution of `initialization`. Then `condition` is evaluated. If `condition` is false, then the statement terminates. If the `condition` is true, the statements in the `body` are executed followed by the `update`. We repeat this process again starting from evaluation of `condition`. This is shown as a flowchart in Figure 6.4.

Note that none of the fields `initialization`, `condition`, `update` or `body` can be empty. If the `condition` is empty, then it is taken as `true`.

The variable named in `initialization` and `update` is customarily called the *control variable* of the loop. As you might expect, `initialization` assigns an initial value to the control variable, and the `update` says how the variable must change from one iteration to the next. As you can see, in our cube table example, the `update` indeed adds 1 to the control variable.

You probably also see why the statement is called a `for` statement. It is because we execute the `body` many times, *for* different values of the control variable.

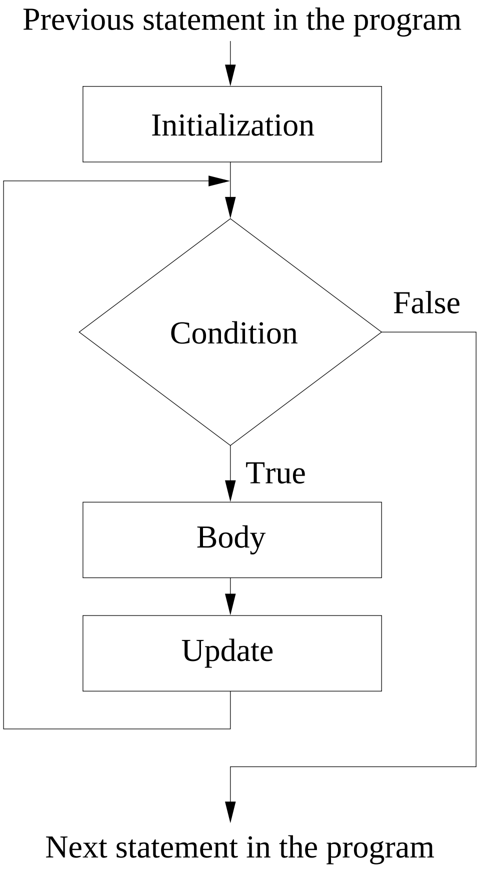

Previous statement in the program
Initialization
Condition
False
True
Body
Update
Next statement in the program

Figure 6.4: For statement execution

6.6.1 Variables defined in `initialization`

As mentioned above, the `initialization` can contain a variable definition, as in our cube-table program. This variable is created during `initialization`, and is available throughout the execution of the `for` statement, i.e. during all the iterations. It is destroyed only when the execution of the `for` statement ends. Thus such a variable cannot referred to outside the `for`. If the value of the variable is useful after the `for` execution is over, then the variable should be defined before the `for` statement, and only initialized in `initialization`.

What if I define a variable `i` in `initialization`, but an `i` has already been defined earlier? So consider the following code.

```
int i=10;

for(int i=1; i<=100; i = i + 1) cout << i*i*i << endl;

cout << i << endl;
```

In this case, we will have shadowing, as discussed in Section 3.8.3. In particular the `i` defined in the first statement will be different from the one defined in the `for` statement, but it will be the same as the one in the last statement! Thus the `for` statement will print a table of cubes as before. The last statement will print 10, because the variable `i` referred to in it is the variable defined in statement 1.

6.6.2 `Break` and `continue`

If a `break` statement is encountered during the execution of `body`, then the execution of the `for` statement finishes. This is exactly as in the `while` statement.

If the `continue` statement is encountered, then the execution of the current iteration is terminated, as in the `while` statement. However, before proceeding to the next iteration, the `update` is executed. After that control continues with the next iteration, starting with checking `condition` and so on.

6.6.3 Style issue

You may well ask: why should we learn a new statement if it is really not needed? Indeed, any program that uses a `for` statement can be rewritten using a `while`, with a few additional variables and assignments.

The reason concerns style. It is much the same as why we speak loudly on certain occasions and softly on others: our softness/loudness help the listener understand our intent in addition to our words. Likewise, when I write a `for` statement, it is very clear to the reader that I am using a certain common programming idiom in which there is a control variable which is initialized at the beginning and incremented at the end of each iteration. If I use either a `while` statement or a `repeat` statement, then the reader does not immediately see all this.

6.6.4 Primality

Our primality program of Section 6.1.2 does indeed have a control variable: the candidate divisor i. Hence it is nicely written as a for loop.

```
main_program{
  int x; cin >> x;

  bool found = false;
  for(int i=2; (i < x) && !found; i++)
    found = found || (x % i) == 0;

  if(found) cout << "Composite.\n";
  else cout << "Prime.\n";
}
```

6.6.5 Natural log by numerical integration

For any real $x > 0$, its natural logarithm $\ln x$ is defined as the number y such that $e^y = x$ where e is Euler's number ($e \approx 2.71828$). We consider how to compute $\ln x$ given x. There are many ways of doing this, we consider a method based on the following relationship:

$$\ln x = \int_1^x \frac{1}{u} du$$

In other words, $\ln x$ is the area under the curve $y = 1/u$ between $u = 1$ and $u = x$. So we can find $\ln x$ if we can find the area!

Well, we cannot compute the area exactly, but we can approximate it. In general suppose we wish to approximate the area under a curve $f(u) = 1/u$ from some p to q. Then we can get an overestimate to this area by considering the area of the smallest axes parallel rectangle that covers it. The height of this rectangle is $f(p)$ (because f is non-increasing) and the width is $q - p$. Thus our required approximation (over estimate) is $(q - p)f(p)$. This is the strategy we will use, after dividing the required area into n vertical strips. Since the curve goes from 1 to x the width of each strip is $w = (x - 1)/n$. The ith strip extends from $u = 1 + iw$ to $u = 1 + (i+1)w$, where we will consider i to be ranging between 0 and $n - 1$ as is customary, rather than between 1 and n. The height of the rectangle covering this strip is $f(1 + iw)$ and hence the area is $wf(1 + iw)$. Thus the total area of the rectangles is:

$$\sum_{i=0}^{n-1} wf(1 + iw) = \sum_{i=0}^{n-1} w\frac{1}{1 + iw}$$

But evaluation of this formula is easily translated into a program! In fact i will naturally serve as a control variable for our for loop. We will take each successive term of the series and add it into a variable area which we first set to 0. The following is the complete program.

```
main_program{
  float x; cin >> x;       // will calculate ln(x)
  int n; cin >> n;          // number of rectangles to use
  float w = (x-1)/n;        // width of each rectangle
  float area = 0;           // will contain ln(x) at the end.
  for(int i=0; i < n; i++)
     area = area + w /(1+i*w);
  cout << "Natural log, from integral: "<< area << endl;
}
```

We note that C++ already provides you a single command `log` which can be invoked as `log(x)` and it returns the value of the natural logarithm. This command uses some code probably more sophisticated than what we have written above, and it guarantees that the answer it returns will be correct to as many bits as your representation (say 24 bits including sign for `float` and 53 bits including sign for `double`). So we can use the command `log` to check how good our answer is. To do this simply add the line

```
  cout << "Natural log, from built-in function: "<< log(x) << endl;
```

before the end of the program given above. This will cause our answer to be printed as well as the true answer, and so we can compare.

It is worth pointing out that there are two kinds of errors in a computation such as the one above. The first is the error produced by the mathematical approximation we use to calculate a certain quantity. For the natural log, this corresponds to the error that arises because of approximating the area under the curve by the area of the rectangles. This error will reduce as we increase n, the number of rectangles. The second kind of error arises because on a computer numbers are represented only to a fixed precision. Thus, we will have error because our calculation of the area of each rectangle will itself not be exact. If we use `float` representation then every number is correct only to a precision of about 7 digits. If you add n numbers each containing an (additive) error of ϵ, then the error in the sum could become $n\epsilon$, assuming all errors were in the same direction. Even assuming that the errors are random, it is possible to show that the error will be proportional to $\sqrt{n}\epsilon$. In other words, if you add 10000 numbers, each with an error of about 10^{-7}, your total error is likely to have risen to about 10^{-5} (if not to 10^{-3}). Thus, we should choose n large, but not too large. The exercises ask you to experiment to find a good choice. Note that you can reduce the second kind of error by representing the numbers in `double` rather than `float`.

Another variation on the method is to approximate the area under the curve by a sequence of trapeziums. This indeed helps. This method, and the more intriguing method based on Simpson's rule are left to the exercises.

Later we will see other methods of computing mathematical functions (including the natural log) which will be based on completely different ideas. The C++ supplied command `log` likely uses one of these other methods.

6.7 Uncommon ways of using `for`

Most often, the `initialization` and `update` in the `for` statement each consists of an assignment to a single variable. However, there are other possibilities too, as we will see in this section.

6.7.1 Comma separated assignments

Here is how we might solve the digit counting problem of Section 6.1.1 using the `for` statement.

```
main_program{
  int n; cin >> n;

  int d, ten_power_d;
  for(d=1, ten_power_d = 10; ten_power_d <= n; d++, ten_power_d *= 10);

  cout << "The number has " << d << " digits." << endl;
}
```

There are two noteworthy features of the `for` statement in the above code. First, the `initialization` and `update` both consist of two assignments separated by a comma. This is allowed. It turns out in C++ the comma is considered to be an operator in such a context, and it merely joins together two assignments!

The second noteworthy aspect is that the above `for` statement has no `body`. This is acceptable.

The above code is very compact, but might be considered tricky by some. The point of to note, of course, is that comma separated assignments can be used as `initialization` and `update` in a `for` statement in general.

6.7.2 Input in `initialization` and `update`

Here is how we could write the mark averaging code using a `for` statement.

```
main_program{
  float nextmark,sum=0;
  float count=0;

  for(cin >> nextmark; nextmark >= 0; cin >> nextmark){
    count++;
    sum += nextmark;
  }
  cout << sum/count;
}
```

We said that `initialization` and `update` in a `for` statement must be expressions; but it turns out that `cin >> nextmark` is an expression! We will discuss what value it returns in Section 8.4.4. But right now the value does not concern us; so you can go ahead and use such input expressions in `initialization` and `update`.

I suspect that some programmers will like this way of writing the program. It is a bit unconventional, however, it does make sense to consider `nextmark` to be a control variable for this program.

6.8 Remarks

Looping is a very important operation in programming. In this chapter we have seen how various problems can be solved using the `while` loop as well as the `for` loop, and earlier we saw the `repeat` loop. For `while` and `for`, there were further variations depending upon whether we used `break`, or replicated code. Later on in the book, we will see even further ways of expressing some of the programs we have seen in this chapter.

As we have indicated, each way of writing loops has some advantages and disadvantages. One may be more readable or less readable, another may avoid duplication of code, and yet another may be less efficient because it does unnecessary work. Another consideration is *naturalness*: does a certain way of writing code more consistent with how you might think about the problem? So the choice of how to express a program is in the end a subjective choice. So you should develop your own taste in this regard.

6.8.1 Nested loops

The `break` statement allows us to break out of only the innermost `while` or `for` statement in which it is contained. Likewise, the `continue` statement causes execution to skip the rest of the `body` of the innermost `while` or `for` statement containing it.

Exercises

1. Write a program that returns the approximate square root of a non-negative integer. For this exercise define the approximate square root to be the largest integer smaller than the exact square root. Your are expected to not use the built-in `sqrt` command, of course.

2. Write a program that reads in a sequence of characters, one at a time, and says whether it contains the sequence of characters 'a', 'b', 'r', 'a', 'c', 'a', 'd', 'a', 'b', 'r', 'a', i.e. the string "abracadabra". Hint: What information do you need to have after reading each character? Try to maintain that.

3. Suppose some code contains some `while` statements. Show how you can replace the `while` statements by `for` statements without changing the output produced by the code.

4. Run the program to compute natural logarithm and check whether it gives better answers for higher values of `n`.

5. Write a program that computes x^n given x and n.

6. Write a program that computes $n!$ given n.

7. Write a program that prints a conversion table from Centigrade to Fahrenheit.

8. Run the program for computing natural log for various choices of n and see how the result varies. For what value of n do you get an answer closest to the `log` function of C++?

9. A more accurate estimate of the area under the curve is to use trapeziums rather than rectangles. Thus the area under a curve $f(u)$ in the interval $[p, q]$ will be approximated by the area of the trapezium with corners $(p, 0)$, $(p, f(p))$, $(q, f(q))$, $(q, 0)$. This area is simply $(f(p) + f(q))(q - p)/2$. Use this to compute the natural logarithm.

10. Simpson's rule gives the following approximation of the area under the curve of a function f:
$$\int_a^b f(x)dx \quad \approx \quad \frac{b-a}{6}\left(f(a) + 4f\left(\frac{a+b}{2}\right) + f(b)\right)$$
Use this rule for each strip to get another way to find the natural log.

11. Suppose we are given n points in the plane: $(x_1, y_1), \ldots, (x_n, y_n)$. Suppose the points are the vertices of a polygon, and are given in the counterclockwise direction around the polygon. Write a program to

    (a) calculate the perimeter of the polygon.

    (b) calculate the area of the polygon. Hint 1: Break the area into small triangles with known coordinates. Then compute the lengths of the sides of the triangles, and then use Heron's formula to find the area of the triangles. Then add up. Hint 2: Break the boundary of the polygon into two parts, an up facing boundary and a down facing boundary. Express the area as the area under these boundaries each considered as functions $f(u)$.

12. Add a "Stop" button to the turtle controller of Section 5.4.1. Modify the program so that it runs until the user clicks on the stop button.

13. Write a program that prints out the digits of a number starting with the least significant digit, going on to the most significant. Note that the least significant digit of a number `n` is simply `n % 10`.

14. Write a program that takes a number `n` and prints out a number `m` which has the same digits as `m`, but in reverse order.

15. A natural number is said to be a palindrome if the sequence its digits is the same whether read left to right or right to left. Write a program to determine if a given number is a palindrome.

16. Write a program that takes as input a natural number x and returns the smallest palindrome larger than x.

17. Write a program that takes a natural number and prints out its prime factors.

18. Let $x_1, \ldots, x_n$ be a sequence of integers (possibly negative). For each possible subsequence $x_i, \ldots, x_j$ consider its sum S_{ij}. Write a program that reads in the sequence in order, with n given at the beginning, and prints out the maximum sum S_{ij} over all possible subsequences.

    Hint: This is a difficult problem. However, it will yeild to the general strategy: figure out what set of values $V(k)$ we need to remember having seen the first k numbers.

When you read the $k+1$th number, you must compute $V(k+1)$ using the number read and $V(k)$ which you computed earlier.

Chapter 7

Computing common mathematical functions

In this chapter we will see ways to compute some common mathematical functions, such as trigonometric functions, square roots, exponentials and logarithms. We will also see how to compute the greatest common divisor of two numbers using Euclid's algorithm. This is one of the oldest interesting algorithm, invented well before computers were even conceived.

The main statement in all the programs of the chapter will be a looping statement. You could consider this chapter to be an extension of the previous, giving more ways in which loop statements can be used.

Some of the material in this chapter requires somewhat deep mathematics. We will state the relevant theorems, and try to explain intuitively why they might be true. The precise proofs are outside the scope of this book.

7.1 Taylor series

Suppose we wish to compute $f(x)$ for some function f, such as say $f(x) = \sin(x)$. Suppose we know how to compute $f(x_0)$ for some fixed x_0. Suppose that the derivative f' of f and the derivative f'' of f' and so on exist at x_0, and we can evaluate these. Then if x is reasonably close to x_0 then $f(x)$ equals the sum of the *Taylor series* of f at x_0. The ith term of the Taylor series is $f^{i\prime}(x_0)(x-x_0)^i/i!$, in which $f^{i\prime}$ is the function obtained from f by taking derivative i times. Thus we have:

$$f(x) = f(x_0) + f'(x_0)(x - x_0) + f''(x_0)\frac{(x-x_0)^2}{2!} + f'''(x_0)\frac{(x-x_0)^3}{3!} + \cdots$$

In the typical scenario, we only compute and sum the first few terms of the series, and that gives us a good enough estimate of $f(x)$. The general theory of this is discussed in standard mathematics texts and is outside our scope. However, you may recognize the first two terms as coming from a tangent approximation of the curve, as shown in Figure 7.1. The value of $f(x)$ equals (the length of) FD. We approximate this by FC, which in turn is FB + BC = EA + (BC/AB)AB = $f(x_0)$ + $f'(x_0) \cdot (x - x_0)$.

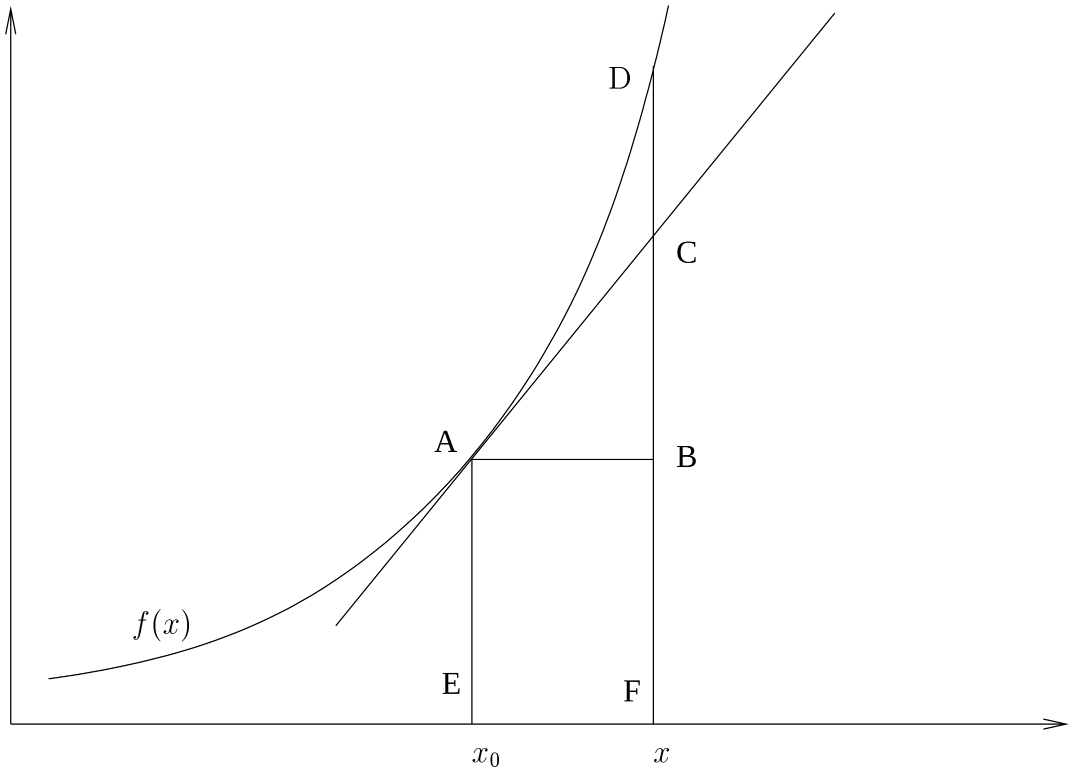


Figure 7.1: Tangent approximation of f at A, $(x_0, f(x_0))$

7.1.1 Sine of an angle

As an example, consider $f(x) = \sin(x)$, where x is in radians. Then choosing $x_0 = 0$ we know $f(x_0) = 0$. We know that $f'(x) = \cos(x)$, $f''(x) = -\sin(x)$, $f'''(x) = -\cos(x)$ and so on. Since $\cos(0) = 1$, we know the exact value of every derivative, it is either 0, 1 or -1. Thus we get

$$\sin(x) = x - \frac{x^3}{3!} + \frac{x^5}{5!} - \frac{x^7}{7!} + \frac{x^9}{9!} - \cdots$$

Here the angle x is in radians. When a series has alternating positive and negative terms, and the terms get closer and closer to 0 for any fixed x, then it turns out that the error in taking just the first k terms is at most the $k+1$th term (absolute value). The kth term of our series is $(-1)^{k+1}x^{2k-1}/(2k-1)!$. Thus if we want the error to be ϵ then we should ensure $x^{2k+1}/(2k+1)! \leq \epsilon$.

We have already seen how to sum series (Section 3.4.2). Here we point out a slightly different way of viewing the problem. Clearly we will need a loop, in the kth iteration of which we will calculate the series to k terms. We must terminate the loop if the last added term is smaller than our target ϵ. We can calculate the kth term t_k from scratch in the kth iteration, but it is useful to note the following relationship:

$$t_k \;=\; (-1)^{k+1}\frac{x^{2k-1}}{(2k-1)!} \;=\; t_{k-1}\left((-1)\frac{x^2}{(2k-2)(2k-1)}\right)$$

provided $k > 1$. If $k = 1$ then $t_k = 1$, of course, and we dont use the above relationship. Thus within the loop we only compute the terms for $k = 2, 3, \ldots$ as needed. Thus our code becomes:

```
main_program{
  double x;  cin >> x;

  double epsilon= 1.0E-20, sum=x, term = x;
  for(int k=2; abs(term) > epsilon; k++){
    // Invariant: on entry term has the value t_{k-1} as discussed.
    // sum has value = sum of k-1 terms
    term *= -x * x /((2*k-2)*(2*k-1));
    sum += term;
  }

  cout << sum << endl;
}
```

The command abs stands for absolute value, and returns the absolute value of its argument.

7.1.2 Natural log

Consider $f(x) = \ln x$, the natural logarithm of x. In the previous chapter we computed this by finding the area under the curve $1/x$. Here we will use the Taylor series. Clearly

$f'(x) = 1/x$. $f''(x) = -1/x^2$ and so on. It is convenient to use $x_0 = 1$. Thus we get:

$$\ln 1 + h = h - \frac{h^2}{2} + \frac{h^3}{3} - \frac{h^4}{4} \cdots$$

A very important point to note for this series is that the series is valid only for $-1 < h \leq 1$. We noted earlier that the Taylor series is valid only if x is close enough to x_0, or equivalently $x - x_0 = h$ is small. For the ln function, we have a precise description of what close enough means: within a unit distance from x_0.

Even so, note that you can indeed use the series to calculate $\ln x$ for arbitrary values of x. Simply observe that $\ln x = 1 + \ln \frac{x}{e}$. Thus by factoring out powers of e we will need to use the series only on a number smaller than 1.

7.1.3 Some general remarks

Note that the Taylor series is often written as

$$f(x_0 + h) = f(x_0) + f'(x_0)h + f''(x_0)\frac{h^2}{2!} + f'''(x_0)\frac{h^3}{3!} + \cdots$$

If we choose $x_0 = 0$, we get the McLaurin series, which is

$$f(x) = f(0) + f'(0)x + f''(0)\frac{x^2}{2!} + f'''(0)\frac{x^3}{3!} + \cdots$$

In general, the terms of the Taylor series increase with x. Thus, it is best to keep x small if possible. For example, suppose we wish to compute sin(100.0). One possibility is to use the previous program specifying 100.0 as input. A better way is to subtract as many multiples of 2π as possible, since we know that $\sin(x + 2n\pi) = \sin(x)$ for any integer n. In fact identities such as $\sin(x) = -\sin(\pi - x)$ can be used to further reduce the value used in the main loop of the program. In fact, noting that the Taylor series for $\cos(x)$ is:

$$\cos(x) = 1 - \frac{x^2}{2!} + \frac{x^4}{4!} - \frac{x^6}{6!} + \cdots$$

we can in fact compute the sine using $\sin(x) = \cos(\pi/2 - x)$ if $\pi/2 - x$ happens to be smaller in absolute value than x.

The proof of the Taylor series expansion is well beyond the scope of this book. However, we have provided some intuitive justification for the first two terms by considering the tangent to approximate the function curve, Figure 7.1.[1]

7.2 Bisection method for finding roots

A root of a function f is a value x_0 such that $f(x_0) = 0$. In other words, a point where the plot of the function touches the x axis. Many problems can be expressed as finding the roots

[1] You might be familiar with the formula $s(t) = ut + \frac{1}{2}at^2$, in which $s(t)$ is the distance covered in time t by a particle moving at a constant acceleration a, with initial velocity u. Note that $u = s'(0)$, $a = s''(0)$ and with this substitution the formula can be written as $s(t) = s(0) + s'(0)t + s''(0)\frac{t^2}{2}$, which resembles the Taylor series in the first 3 terms.

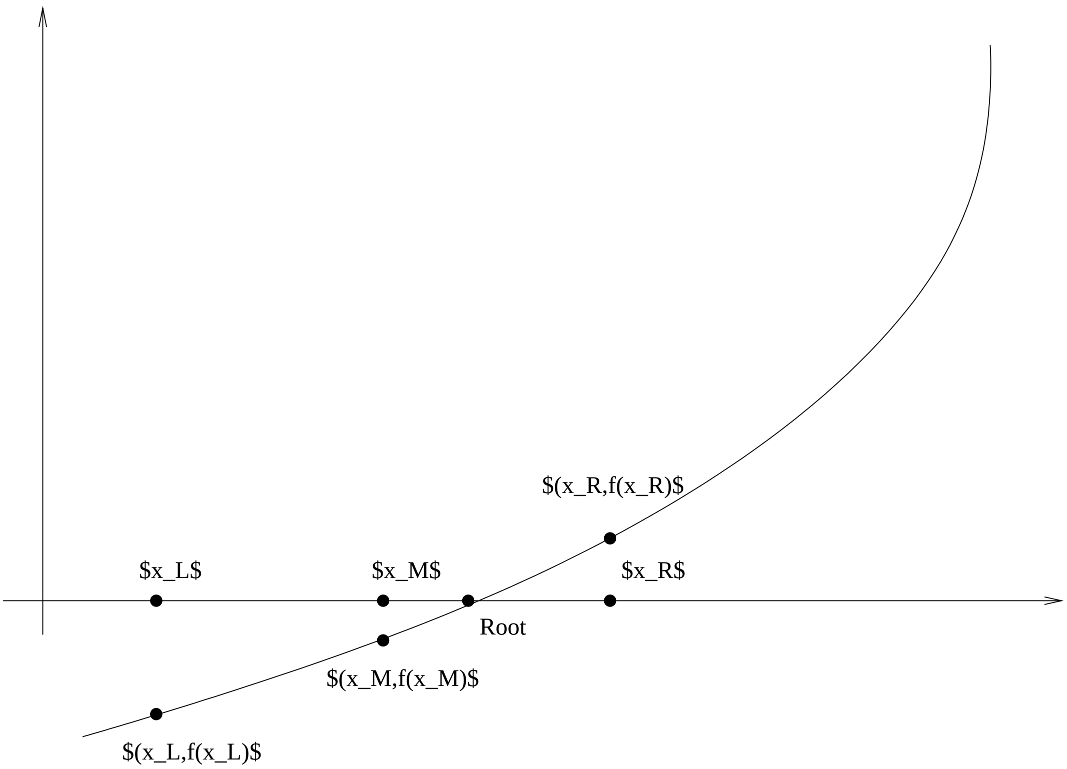


Figure 7.2: Bisection method. Next we will have $x_L = x_M$.

of an equation. For example, suppose we want to find the square root of 2. Then instead we could ask for the roots of the polynomial $f(x) = x^2 - 2$. Clearly, if $f(x) = 0$ then we have $x^2 - 2 = 0$, i.e. $x = \pm\sqrt{2}$ and this would give us the square root of 2. So finding roots is a very important mathematical problem.

In this section, we will see a very simple method for finding roots *approximately*. The method will require that (a) we are given values $x_L \leq x_R$ such that $f(x_L)$ and $f(x_R)$ have opposite signs, (b) f is continuous between x_L and x_R. These are fairly minimal conditions, for example for $f(x^2) = x^2 - 2$ we can choose $x_L = 0$ giving $f(x_L) = -2$, and $x_R = 2$ (or any large enough value), giving $f(x_R) = 2$. Clearly x_L, x_R satisfy the conditions listed above.

Because f is continuous, and has opposite signs at x_L, x_R, it must pass through zero somewhere in the (closed) interval $[x_L, x_R]$. We can think of $x_R - x_L$ is the degree of uncertainty, (or maximum error) in our knowledge of the root. Getting a better approximation merely means getting a smaller interval, i.e. x_L, x_R such that $x_R - x_L$ is smaller. If the size of the interval is really small, we can return either endpoint as an approximate root. So the main question is: can we somehow pick better x_L, x_R given their current values.

A simple idea works. Consider the interval midpoint: $x_M = (x_L + x_R)/2$. We compute x_M and find the sign of $f(x_M)$. Suppose the sign of $f(x_M)$ is different from the sign of $f(x_L)$. Then we know can set $x_R = x_M$. Clearly the new values x_L, x_R satisfy our original requirements. If the sign of x_M is the same as the sign of x_L, then it must be different from the sign of x_R. In that case (see Figure 7.2) we set $x_L = x_M$. Again the new values of x_L, x_R satisfy our 2 conditions. Hence in each case, we have reduced the size of the interval, and thus reduced our uncertainty. Indeed if we want to reduce our error to less than some ϵ, then we must repeat this process until $x_R - x_L$ becomes smaller than ϵ. Then we would know that they are both at a distance at most ϵ from the root, since the root is inside the interval $[x_L, x_R]$.

The code is then immediate. We write it below for finding the square root of 2, i.e. for $f(x) = x^2 - 2$.

```
main_program{              // find root of f(x) = x*x - 2.
  float xL=0,xR=2; // invariant: f(xL),f(xR) have different signs.
  float xM,epsilon;
  cin >> epsilon;
  bool xL_is_positive, xM_is_positive;
  xL_is_positive = (xL*xL - 2) > 0;
  // Invariant: x_L_is_positive gives the sign of f(x_L).

  while(xR-xL >= epsilon){
    xM = (xL+xR)/2;
    xM_is_positive = (xM*xM -2) > 0;
    if(xL_is_positive == xM_is_positive)
      xL = xM;  // does not upset any invariant!
    else
      xR = xM;  // does not upset any invariant!
  }
  cout << xL << endl;
}
```

7.3 Newton Raphson Method

We can get a faster method for finding a root of a function f if we have a way of evaluating $f(x)$ as well as its derivative $f'(x)$ for any x. To start off this method, we also need an initial guess for the root, which we will call x_0. Often, it is not hard to find an initial guess; indeed in the example we will take, almost any x works as the initial guess.

In general, the Newton-Raphson method takes as input a current guess for the root, say x_i. It returns as output a (hopefully) better guess, say x_{i+1}. We then compute $f(x_{i+1})$, if it is close enough to 0, then we report x_{i+1} as the root. Otherwise, we repeat the method with x_{i+1} to get, hopefully, an even better guess x_{i+2}.

The process of computing x_i given x_{i+1} is very intuitive. We know from Section 7.1 that $f(x) \approx f(x_i) + f'(x_i) \cdot (x - x_i)$, assuming $x - x_i$ is small. In this equation we could choose x to be any point, including the root. So let us choose x to be the root. Then $f(x) = 0$. Thus we have $0 \approx f(x_i) + f'(x_i) \cdot (x - x_i)$. Or in other words, $x \approx x_i - \frac{f(x_i)}{f'(x_i)}$. Notice that the right hand side of this equation can be evaluated. Thus we can get an approximation to the root! This approximation is what we take as our next candidate.

$$x_{i+1} = x_i - \frac{f(x_i)}{f'(x_i)} \tag{7.1}$$

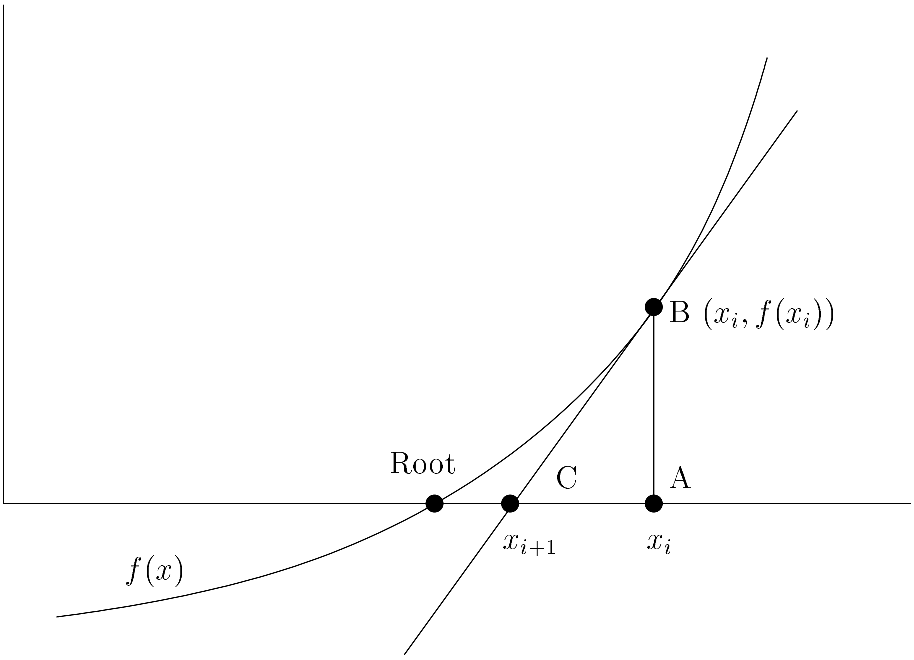


Figure 7.3: One step of Newton-Raphson

That is all there is! Figure 7.3 shows what happens graphically. The point A with coordinates $(x_i, 0)$ represents our current estimate of the root. We draw a vertical line from A up to the function taking us to the point B, having coordinates $(x_i, f(x_i))$. At B we draw a tangent to the function f, and the tangent intersects the x axis in point C. If we consider the tangent to be a good approximation to f, then the root must be point C. Indeed, we take the x coordinate of C to be our next estimate x_{i+1}. Thus we have

$$x_{i+1} = x_i - AC = x_i - \frac{AB}{AB/AC} = x_i - \frac{f(x_i)}{f'(x_i)}$$

which is what we obtained earlier arguing algebraically. At least in the figure, you can see that our new estimate is better, indeed the point C has moved closer to the root as compared to the point A. It is possible to argue formally that if x_i is reasonably close to root to start with, then x_{i+1} will be even closer. Indeed, in many cases, it can be shown that the number of bits of x_i that are correct essentially double in going to x_{i+1}. Thus a very good approximation to the root is reached very quickly. The proof of all this is not too hard, at least for special cases, but beyond the scope of this book.

We now show how the Newton-Raphson method can be used to find the square root of any number y. As with the bisection method, we must express the problem as that of finding the root of an equation: $f(x) = x^2 - y$. We also need the derivative, and this is $f'(x) = 2x$. Next we need an initial guess. The standard idea is to make an approximate plot of the function, and choose a point which appears close to the root. In this case it turns out that almost any initial guess is fine, except for 0. So for simplicity, we choose $x_0 = 0$. The update rule, $x_{i+1} = x_i - \frac{f(x_i)}{f'(x_i)}$ in this case becomes

$$x_{i+1} = x_i - \frac{x_i^2 - y}{2x_i} = \frac{1}{2}(x_i + \frac{y}{x_i})$$

So we are ready to write the program. The basic idea is to maintain a variable `xi` representing the current guess. We will update `xi` in each iteration using the above rule, and initialize `xi` to 1.

```
main_program{
  double xi=1, y;  cin >> y;
  repeat(10){
    xi = (xi + y/xi)/2;
  }
  cout << xi << endl;
}
```

This program will run a fixed 10 iterations, and calculate the estimates $x_1, x_2, \ldots, x_{10}$, starting with $x_0 = 1$. But we can also run a number of iterations depending upon how much error we wish to tolerate.

This is slightly tricky. If the actual root is x^*, then the error in the current estimate x_i is $|x_i - x^*|$. Indeed, if we exactly knew the error, i.e. the value $v = |x_i - x^*|$, we could directly compute the root by noting that $x^* = x_i \pm v$. So we need to make an estimate for the error. A common estimate is $f(x_i)$. Indeed, $f(x_i)$ is the vertical distance of the point $(x_i, 0)$ to the curve f whereas the exact error, $x_i - x^*$ is the horizontal distance of the point $(x_i, 0)$ to the curve. Indeed, when the vertical distance becomes 0, so would the horizontal. So our program can terminate when the error estimate $f(x_i) = x_i^2 - y$ becomes small, i.e. when `xi*xi-y` becomes smaller than some threshold

```
main_program{
  float y; cin >> y;
  float xi=1;
  while(abs(xi*xi - y) >0.001){
    xi = (xi + y/xi)/2;
    cout << xi << endl;
  }
}
```

In the above code we have used the built-in function `abs` which returns the absolute value of its argument.

7.4 Greatest Common Divisor

The inputs for this problem are positive integers m, n. The output required is the largest integer dividing both.

The algorithm for this, as taught in primary schools, is to factorize both the numbers, and then the greatest common divisor (GCD) is the product of the common factors. Another possibility is to go with the specification: examine the numbers between 2 and $\min(m,n)$, and find the largest one that divides both. This will work, but is slower than the primary school method.

Euclid's algorithm is much faster than both these methods. The starting point for it is a relatively simple observation: if d is a common divisor of positive integers m, n then it is a common divisor also of $m-n, n$, assuming $m > n$. The proof is simple: Since d divides m, n we have $m = pd, n = qd$, for integers p, q. Thus $m - n = (p-q)d$, and hence d divides $m - n$ also. By a similar argument you can also prove the converse, i.e. if d is a common divisor of $m-n, n$, then d is a common divisor of m, n also.

Thus we have shown that every common divisor of m, n is also a common divisor of $m-n, n$, and vice versa. But then it means that the set of common divisors of m, n is identical to the set of common divisors of $m-n, n$. Thus the greatest in the first set must be the greatest in the second set, i.e. $GCD(m,n) = GCD(m-n,n)$.

The last statement has profound consequences. It should be read as saying: if you want the GCD of m, n, you may instead find the GCD of $m-n, n$ assuming $m > n$. This could be considered progress, because intuitively, you would think that finding the GCD of smaller numbers should be easier than finding the GCD of larger numbers.

Let us take an example. Suppose we want to find the GCD of 3977, 943. Thus we have $GCD(3977, 943) = GCD(3977-943, 943) = GCD(2034, 943)$. But there is no reason why we should use this idea just once: we can use it many times. Thus we get $GCD(2034, 943) = GCD(2091, 943) = GCD(1148, 943) = GCD(205, 943)$. At this point you might realize that we can subtract all multiples in one shot, and the result is simply the remainder when dividing the original number 3977 by 943. Thus we could more directly have written $GCD(3977, 943) = GCD(3977\%943, 943) = GCD(205, 943)$.

Because GCD is a symmetric function we can subtract multiples of m just as well as n. Thus $GCD(205, 943) = GCD(205, 943\%205) = GCD(205, 123)$. This further simplifies: $GCD(205, 123) = GCD(205\%123, 123) = GCD(82, 123) = GCD(82, 123\%82) = GCD(82, 41)$. At this point if we try to apply our rule we get $82\%41 = 0$, i.e. the smaller of the numbers divides the larger, and so it must be the GCD. Thus we have obtained, overall, that GCD(3977,943)=41.

We can summarize the ideas above into a simple theorem.

Theorem 1 (Euclid) *Suppose m, n are positive integers. If $m\%n = 0$, then $GCD(m,n) = n$. Otherwise $GCD(m,n) = GCD(m\%n, n)$.*

This is enough to write a program. In the program, we will want to divide the larger number by the smaller. So we will keep our numbers in variables `Large` and `Small` and ensure all along that `Large` always will have the larger and `Small` the smaller number.

```
main_program{
  int Large, Small, Remainder;
  cout << "Enter the larger number: ";
  cin >> Large;
  cout << "Enter the smaller number: ";
  cin>> Small;
  while(true){
    Remainder = Large % Small;  // Comment 1: Remainder < Small
    if (Remainder == 0) break;
    Large = Small;
    Small = Remainder;          // Comment 2: Small < Large
  }
  cout << "The GCD is: " << Small << endl;
}
```

We will prove the invariant that Large will contain a larger value than the value in Small on any entry into the loop. We will assume that the user followed our instruction, and hence the invariant will be true on the first iteration. For any subsequent iteration, note that after the first statement of the loop, Remainder will have a smaller value than Small, by definition of division (Comment 1 in the code). The last two statements of the loop respectively assign the values of Small and Remainder (decreasing order) to Large and Small. Thus at the end of the loop Large will have a larger value than Small. This relationship will remain unchanged to the beginning of the next iteration.

It should be intuitively clear that our program will terminate and work correctly. We now observe it formally. We chose the new values of Large and Small so that their GCD remains invariant from one iteration to the next. Further, the value of Large in the next iteration is the current value of Small, which is known to be smaller than the current value of Large. Hence in each iteration, the value of Large decreases. Since it cannot drop below 1, the number of iterations cannot be more than the value of Large typed in by the user at the beginning of the program. When the program terminates, we further know that Remainder == 0, i.e. Large % Small == 0. But then the GCD must be Small, which is indeed what we print. Thus we have established correctness as well as termination.

Actually, we can argue that the number of iterations are much fewer than the original value of Large. Let L_i, S_i denote the value of Large and Small respectively at the beginning of the ith iteration. Let R_i denote the value of Remainder calculated in the ith iteration. Then we know:

1. $L_i = qS_i + R_i \geq S_i + R_i$.
2. $L_{i+1} = S_i$, $S_{i+1} = R_i$. This follows by considering the assignments we make at the end of the loop.

Thus we have $L_i \geq S_i + R_i = L_{i+1} + S_{i+1} \geq 2L_{i+2}$. Thus we have established that the value of Large drops by a factor at least 2 in 2 iterations. Thus the number of iterations is at most $2\log_2 L_0$, where L_0 is the value of Large as typed in by the user.

We will finally note that our program runs correctly even if the user disregards our instructions and types in the smaller number first and larger second. This changes the invariants and the analysis slightly, and you are asked about this in the Exercises.

7.5 Summary

This chapter has introduced several new ideas, though no new programming langauge features.

7.5.1 Mathematical ideas

We saw some general techniques for computing mathematical functions. We saw that if the function and its derivatives are easy to evaluate for some values of the argument, then the Taylor series of the function can often be used to give an algorithm that evaluates the function for all values of the argument.

We also saw that computation of a function f could be expressed as the problem of finding the roots of another function g. Roots can often be found using various methods. These methods require that we should be able to evaluate g and possibly its derivatives at arbitrary points.

7.5.2 Programming ideas

The first idea was that the values required in each iteration of the loop need not be calculated afresh, they could be calculated from values calculated in the preceding iterations. We saw the use of this idea in the calculation of $\sin x$.

The second important idea was that some properties of certain variables may remain unchanged over the iterations of a loop. We saw for example that $GCD(m, n)$ remained the same at the beginning of each iteration of the loop in our program. Such a property that remains the same at a given point in the loop is commonly called a *loop invariant*. The notion of an invariant is important in reasoning about programs.

The final important idea is that of a *potential function*. We argued that the GCD program must terminate because the value of `Large` had to decrease in each iteration of the loop, and the value could never drop below zero. It is customary to refer to such a quantity as a potential function for the loop. If we can argue that the potential must steadily drop but cannot drop below a certain limit, then the only way this can happen is if the loop stops after a finite number of iterations. The name arises from Physics, where customarily the particles (or systems) have a tendency to go from high potential (energy) to low.

Exercises

1. Write a program to find $\ln x$ for arbitrary x using the Taylor series. Check your answer by using the built in *log* command.

2. Write down the Taylor series for $f(x) = e^x$, noting that $f^{i'}(x) = e^x$. It is convenient to expand around $x_0 = 0$, i.e. consider the MacLaurin series. This series is valid for all values of x, however, it is a good idea to use it on as small values of x as possible. Write a program to compute e^x, and check against the built in command `exp`.

3. Children often play a guessing game as follows. One child, Kashinath, picks a number between 1 and 1000 which he does not disclose to another child, Ashalata. Ashalata asks questions of the form "Is you number between x and y?" where she can pick x, y as she wants. Ashalata goal is to ask as few questions as possible and guess the number that Kashinath picked. Show that Ashalata can guess the number correctly using at most 10 questions. Hint: cast it as a root finding problem.

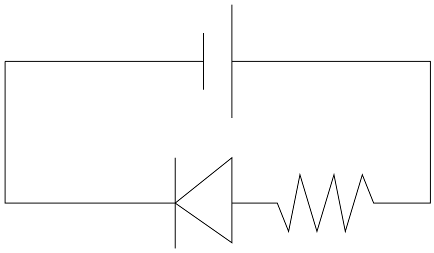


Figure 7.4: Diode circuit

4. Add checks to the GCD code to ensure that the numbers typed in by the user are positive. For each input value you should prompt the user until she gives a positive value.

5. Suppose the user types in the smaller number first and the larger number second, in response to the requests during the execution of the GCD program of Section 7.4. Show that the correct answer will nevertheless be given. State how the invariants and the analysis of the number of iterations will change.

6. Write a program to find arcsin(x) given x.

7. Consider a circuit in which a voltage source of $V_{CC} = 1.5$ volts is applied to a diode and a resistance of $R = 1$ Ohm connected in series, Figure 7.4. The current I through a diode across which there is a potential drop of V is

$$I = I_S(e^{V/(nV_T)} - 1)$$

where I_S is the reverse saturation current of the diode, V_T is the thermal voltage which is about 25 mV at room temperature (300 degrees Kelvin), and n is the ideality factor. Suppose the diode we are using has $n = 1$ and $I_S = 30$ mA. Write a program that finds the current. Use your program to also find the current when the voltage source is reversed.

8. Consider the problem of finding the roots of $f(x) = x^3 - x/2 + 1/4$. See what happens using the Newton-Raphson method for guesses for the initial value. In particular, try $x_0 = 1$ and $x_0 = 0.5$. Can you solve this using the bisection method?

Chapter 8

Testing and Debugging

Only the paranoid survive.
Title of book by Andy Grove, co-founder, Intel Corp.

Writing a program is often a frustrating experience. You write a program and then you test it with different inputs. Quite likely, you discover that it does not work correctly for some inputs. Then you perhaps realize your mistake and so you modify your program. Sometimes this process seems to go on for ever – you start thinking: my program is *clearly* correct, is it possible that the computer is making a mistake? But of course, computers make mistakes so rarely that you can assume that they never make mistakes. And to add insult to injury, it often happens that eventually when you discover why the program did not work, you realize that it was a fairly “stupid” mistake, something that you never thought you would be capable of, for which you want to kick yourself. So how then, should you work towards getting a correct program? How do you avoid mistakes, big, small or stupid?

In this chapter, we make a few suggestions regarding the general process of program development. For the seemingly simple programs that you have been asked to write so far, perhaps an intuitive process might be fine – each programmer does what he/she *feels* is the right thing to do. However, there are certain simple principles that could be followed which might help in getting correct programs faster and without getting frustrated. We present some suggestions for each stage of the process: from the time you start thinking about the problem all the way to testing your program.

We suspect that the temptation to write programs *intuitively* hoping that they will *just work* is too strong. But especially if you find that your intuitive style is making you spend too much time correcting the errors in the programs you have written *spontaneously*, at least then consider the ideas given below.

8.1 Clarity of specification

The first step in writing a correct program is to clearly know what you want the program to do. This might sound obvious, but most often, programs dont work because the programmer did not clearly consider what needs to happen for some tricky input instance. How can you be sure that you completely *know* what the program is expected to do, that you have considered all the possibilities? The best way for doing this is to *write* down the specifications very clearly, in precise mathematical terms if possible. By specifications we mean a characterization of the output in terms of the input. The specification does not state how the output is to be actually produced; it merely says: if the output satisfies these conditions then it is to be certified as correct. It is a good idea to write the specification as a comment in your program, and also to use the same variable names in the specification as in your program.

Let us take an example. What are the specifications for the program to count digits of a number? We think that we understand decimal numbers, which we indeed do. But such intuitive understanding does not constitute a specification. The intuitive understanding must be explained in more precise terms. Here is the specification we used earlier.

Input: A non negative integer n.

Output: Positive integer d such that d is the smallest integer satisfying $10^d > n$.

This is a very good specification because it gives the precise conditions that we want the output to satisfy. Further, the conditions are *checkable*: given a solution d^* to the problem you can quickly check the conditions, i.e. check if $10^{d^*} > n$ and whether d^* is smallest, i.e. that $10^{d^*-1} \not> n$ provided $d^* - 1$ is a positive integer.

It is customary in writing specifications to state conditions such as *smallest/largest ... satisfying ...* . Formulating specifications in this manner requires some practice. Also a lot of care is needed. Should the condition be $10^d > n$ or $10^d \geq n$? You may consider these questions to be tedious, but if you cannot answer them correctly while writing the specification, you are unlikely to write the program correctly. You may be making the same mistake in your program! Writing the specification is tedious if there are many cases, e.g. say you want to print the day of a week given the date. But in such cases it is imperative that you write down the specifications in complete detail, just to make sure you understand all the nuances in the problem.

As an exercise, try writing specifications for the following problem. Suppose you are given n points in the plane, $p_1, \ldots, p_n$. Find the smallest circle that contains all the points. It might be tempting to rewrite just what is stated in the problem statement:

Input: n points $p_1, \ldots, p_n$ in the plane.

Output: Smallest circle that contains all points.

This is not a good specification because it does not even give a representation for the output. The specification should at least specify what we mean when we say "smallest circle", and what it means for a circle to "contain a point". Here is a better specification of the output.

Output: Point C and a number R such that the distance between the point C and each point p_i is at most R.

This is a good specification, it is understood how a point is represented: it is represented as a pair of numbers, say the x, y coordinates. Note that this specification is only partially checkable: given a proposed solution C^*, R^*, we can check whether all points p_i are within distance R^* from point C^*. However, there is no easy way to check that R^* is smallest, for all choices of C^*.

8.1.1 Input and output examples

Along with writing the specifications, you should construct sample input instances, and work out what output you want for those. For the digit counting program, the input and output are both single numbers, and so this is rather trivial. But even here, it is a good idea to write down specific numerical examples. So if you write input instance: 34, output: 2, check whether this agrees with the abstract specification that you have written. Is 2 indeed the smallest number such that $10^2 > 34$? These may sound like trivial checks, but your program can go wrong because of trivial mistakes, and so such checks are useful.

8.2 Hand execute the program

A fundamental skill that you must have is to pretend you are the computer and execute the program. You *must* do this for small programs and simple inputs. Going back to the digit counting program, you should be able to precisely state how the program executes, when given some small number, say 34, as input. Especially if you find that your program is not working correctly, hand execute it on small inputs.

8.3 Assertions

The import of the previous discussion is: you should know how your program executes. The more you know your program, the less it is likely to have bugs.

Your knowledge of the program is often of the form: "at this point in the program, I expect the value of this variable to be 0". Or you may say, I expect the user to type in a positive value at this stage. Why not actually check these expectations during execution? Yes, making the check will require some execution time, however, if your program is not running correctly, it might well be because something that you confidently expect is not actually happening.

C++ contains a facility which makes it easy to check your expectations and produce error messages if the expectations are incorrect. Suppose you express a certain condition `condition` to hold at a certain point in your program, you simply place the statement

```
assert(condition);
```

Here `condition` must be a boolean expression. When control reaches this statement, `condition` is evaluated, and if it is false an error message is given, typically stating "Assertion failed", and the line number and the name of the program file containing the assertion is also given. If the expression `condition` is true (as you expected it to), then nothing happens and the control just moves to the next statement.

To use `assert` you must include the line

```
#include <assert.h>
```

at the top of your file. You can also instead include `<cassert>` if you wish.

Pre and post conditions as will be discussed in the next chapter are good candidates for using assertions. Here is another simple use. Suppose you know that a certain variable v will only take values 1 or 2. The you might originally have written:

```
...
if(v == 1) ...
else if(v == 2) ...
...
```

You might not write the else statement following the if else thinking that it is not necessary. But "the statement is not necessary" is only your expectation. If the value of the variable v has arisen after a complicated calculation, or after input it from the user, it is conceivable that something might have gone wrong in your deduction or that the user is not following instruction. In both cases, if you are debugging your program, then you want to be sure that your "confident deduction" is actually correct, or that the user is following instructions, before you start suspecting the rest of the program. So you could actually make a check by writing an assertion.

```
...
if(v == 1) ...
else if(v == 2) ...
else assert(false);  // executed only if v is not 1 or 2.
...
```

8.4 Testing

After you develop the program, you will test it on some input instances. What instances do you choose? First run it on the instances for which you already hand executed the program. You may think this is a waste of time: but remember, everybody is capable of making stupid mistakes, and these might be caught on the simplest input instances.

After the small, simple input instances, try some bigger or more complex input instances. For this there are a few strategies to consider. One idea is to generate what you think might be "usual" instances which somehow you think might be "common in practice". For example, for the covering circle problem, the instance in which all points are randomly placed in the plane is perhaps more common than the instance in which they are all collinear. It is possible that for your problem (say counting digits) there is no notion of "common in practice". So for this you can think of using *random* input values. You may wonder how you can feed *random* numbers to a computer. We will discuss this in Section 8.6.

But in addition to random input values, you can try to think about which input values might be *difficult* for the program. The notion of *difficult* is of course informal. But here is how you might consider certain inputs more *interesting*, say for the digit counting problem. If you look at the number of digits d as a function of the input n, you will see that d changes at powers of 10. At 9 the value is 1, but it goes up to 2 at 10. The value is 3 at 999 but goes up to 4 at 1000. So you might want to pay more attention to these values: perhaps the program has to be "keenly attentive" and distinguish between 999 and 1000 (even though they are consecutive), but not between 578 and 579 (which are also consecutive). So checking the inputs 999, 1000 and so on might be more likely to show up the errors, than say checking 578 or 579. Another case of course is to check for the smallest and largest input values allowed. In case of digit counting 0 is the smallest value allowed, and whatever the largest value allowed is for representing n on your computer.

A related case is programs that take variable size inputs. For example, in the mark averaging program (Chapter 6), the number of marks to be given was not specified beforehand, we entered 200, an impossible mark value, to indicate that no more marks would be entered. Does your program run correctly if only one mark is given? What if no marks are given i.e. the first value typed in is itself 200? These are questions you should have thought about clearly when you wrote the program.

Finally, one more suggestion can be given. In general your program will contain `if` statements, i.e. the execution will not follow the same path for every input instance. If so, you should try to figure out as many values as possible for which different statements in your program will need to execute – and test the program on those input values. For example, if you are writing a program to print the day of the week given the date, you should surely test using a 29th February in a leap year – quite likely there will be separate code which checks for this and you want to know that that code runs correctly.

This raises an important issue: you may often want to first check in your program that a valid input is being given by the user. For example, some user might report that your program does not return consecutive days of the week for the dates 28/2/2011 and 29/2/2011, and is therefore wrong. You may unwittingly believe the user and start hunting for the bug, which does not exist!

8.4.1 Automated testing: input/output redirection

If you are debugging a program, then on each run you will have to supply the input. If the input is long it will take a lot of time and effort to type it in. In such cases, you can type in all the input that you want to supply into a file. Say the file is called `input.txt`. Then when you run your program, you can *redirect* the `cin`, the standard input stream, to take input from this file rather than from the keyboard. To do this, instead of typing `./a.out`, you should instead type

```
% ./a.out <input.txt
```

Then the program executes and all statements `cin >> ..` will take input from `input.txt`. You can create multiple input files and in successive executions redirect the input from those files. This way, you need not type the input again for each run. This will reduce your effort and also your frustration.

Note that the standard output stream, `cout`, can also be redirected. If you write

```
% ./a.out >filename
```

then the output will be sent to the file `filename` instead of being shown on the screen. You can examine the file `filename` at your leisure. This is important especially if your output is very long.

8.4.2 File I/O

Instead of redirecting standard input or standard output, you can read or write files too. For reading files you first need to insert the line

```
#include <fstream>
```

at the beginning of the file, along with other lines you may have such as `#include <simplecpp>`. Once you have this you can create a so-called input stream, by writing:

```
ifstream myinfile("input.txt");
```

This creates a variable called `myinfile` in your program, which is a stream, from which you can read values. The quoted name inside the parentheses tells where the stream will get its values: in this case, the statement says that the values will come from the file `input.txt` which must be present in the current working directory. After this, you can treat the name `myinfile` just as you treat `cin`, and you can write statements such as `myinfile >> n;` which will cause a whitespace delimited value to be read from the file associated with myinfile into the variable `n`.

In a similar manner you can write

```
ofstream myoutstream("output.txt");
```

which will create the variable `myoutstream`, of type `ofstream`, and associated with the file `output.txt`. You can treat `myoutstream` just like `cout`, i.e. you can write values to it using statements such as `myoutstream << n;`. The values will get written to the associated file, in this case `output.txt`, which will get created in the current working direcory.

Here is a program which takes the first 10 values from a file `squares.txt` which is expected to be present in your current directory, and copies them to a file `squarecopy.txt`, which will get created.

```
#include <simplecpp>
#include <fstream>

main_program{
  ifstream infile("squares.txt");
  ofstream outfile("squarecopy.txt");

  int val;
  repeat(10){
    infile >> val;
    outfile << val << endl;
    cout << val << endl;
  }
}
```

The values are also printed out on `cout` which means they will also appear on the screen (unless you redirect standard out during execution). Notice that we have chosen to enter an end of line after each value, while printing to `outfile` as well as `cout`.

8.4.3 End of file and reading errors

The variables which take the value ifstream as well as the variable cin behave in a strange but predictable manner, if there is an error while reading, or if the file ends. By an error in reading, we simply mean something such as: you have asked for an integer value, say as you did for the statement infile >> val; above, and the actual value present in the file or typed in in case of cin was not numerical. By end of file we mean that you asked for a value but the file has already ended. This would happen in the program given above if, for example, the file squares.txt contained fewer than 10 values. In the case where cin represents the keyboard, the user can type control-d, i.e. type the letter d while the control key is depressed, and that will be treated by the program as end of the file.

If either a reading error or an end of file occurs, then the concerned stream variable takes the value false. So in fact you can check if either of these conditions happened after each reading operation. Here is a modification to the previous program which makes this check.

```
main_program{
  ifstream infile("squares.txt");
  ofstream outfile("squarecopy.txt");

  int val;
  repeat(14){
    infile >> val;
    if(!infile){
      cout << "Reading error or end of file.\n";
      break;
    }
    outfile << val << endl;
    cout << val << endl;
  }
}
```

8.4.4 Input-Output expressions

Finally, we note that in C++ the phrase infile >> value causes a value to be read from infile into the variable value, and in addition itself is an expression that has a value: the value of the expression is the value of the variable infile. This should not come as a surprise to you, this is in fact the reason you can write statements such as cin >> a >> b; which you should really read as (cin >> a) >> b; where the first expression causes a value to be read into a, and then the expression evaluates to cin, from which another value is read into b.

This fact allows us to write some rather compact loops. Here is a program that merely copies all values from the file squares.cpp to the file squarecopy.cpp, without knowing how many values there were.

```
#include <simplecpp>
#include <fstream>
```

```
main_program{
  ifstream infile("squares.txt");
  ofstream outfile("squarecopy.txt");

  int val;
  while(infile >> val){      // file read in the loop test
    outfile << val << endl;
    cout << val << endl;
  }
}
```

The reading happens in the loop test, and if there is an error or end of file, the reading expression returns false, and the loop ends. Thus the above program will read till there is either an error or the end of the file, and each value read will be printed on `cout` as well as on `outfile`.

8.4.5 Assert with file reading

Ideally, after every read operation from any stream, you should check that the file did not end unexpectedly, nor there was any error. If the `infile` is a stream, then after a statement such as `infile >> val;`, you might find it convenient to write

```
assert(infile);
```

This will print an error message if an end of file was detected, or a reading error happened. Note that C++ does not give an error message by itself if there is an error, it will happily continue, but merely return junk values on subsequent reads!

If you have prepared test files which you plan to use as input to test your programs, we strongly recommend that you use assertions after each read (putting the read in a loop test is equivalent). This way you know that your program is giving the wrong answers because you are giving it wrong data, and not because there is anything necessarily wrong with the program.

8.5 Debugging

Suppose you follow the above directions and are generally very careful, and yet things go wrong: your program produces an answer different from what you expect. What do you do?

The most natural response is to try and find out when the program starts behaving differently from what you expect. For this, you can print out the values of the important variables at some convenient halfway point, and check if the values are as you might expect. If they are not then place print statements for an earlier point. If the values are as you expect at the halfway point, then clearly the computer is doing something unexpected later than the halfway point, and so you put print statements to check the values at a later point in the execution. By examining the values in this manner, you try to get to a single statement until which the values are as you expect, but after which the values are different. At this point, you are usually in a position to determine what is going wrong.

8.6 Random numbers

C++ provides you with the function `rand` which takes no arguments which returns a *random* number. This statement should puzzle you – a computer is an orderly deterministic machine, indeed we did not say anything about randomness in our discussion of computer hardware (Chapter 2). How can then a computer generate random numbers?

Indeed, a computer does not generate truly random numbers. Instead, a computer merely generates successive numbers of a perfectly deterministic computable sequence whose elements *seem* to be resemble a sequence which could have been generated randomly. Such sequences and their elements are said to be *pseudo-random*. Indeed a simple example is the so called linear congruential sequence, given by say, $x_i = a \cdot x_{i-1} + b \bmod M$, where a, b, M are suitably chosen integers. Say we choose$a = 37$, $b = 43$, $M = 101$. Then starting with $x_0 = 10$, the next few terms are: 9, 73, 17, 66, 61, 78, 0, 43, 18, 2, 16. Perhaps you will agree informally that this sequence looks random, or at least more random than the sequence 0, 1, 2, 3, 4 and so on. It is possible to formalize what *pseudo-random* means, but that is outside the scope of this book. So we will just assume that pseudo-random merely gets the best of both worlds: it is a sequence that can be generated by a computer, but can be considered to be random for practical purposes. There is also another convenient property which we will see in a minute.

Functions such as `rand` which return (pseudo) random numbers do use the general idea described above: the next number to be returned is computed as a carefully chosen function of the previous. So the exact sequence of numbers that we get on successive calls to `rand` depends upon how we started off the sequence, what x_0 we chose in the example above. This first number of the sequence is often called the *seed*. C++ allows you to fix the seed by calling another function `srand` which takes a single integer argument which will be used as the seed for the subsequent sequence. To use `rand` and `srand`, you would normally need to include the line

```
#include <stdlib.h>
```

But this is not needed if you are including `<simplecpp>`.

A call `rand()` returns an `int` in the range 0 to `RAND_MAX`. This name is defined for you when `<stdlib.h>` is included. You can consider the returned value to be *uniformly distributed*, i.e. the value is equally likely to be any integer between the specified limits.

Finally, an important point about pseudorandom sequences. The sequence you get when you fix the seed is always the same. This is a desirable property if you will use it to generate input data. This is for the following reason. Suppose your program is not working correctly for certain (randomly generated) data. Say you modify the program and you wish to check if it is now correct. Had the data been truly random, it would be unlikely that the same sequence would get generated during the execution. However, since you use a pseudo random sequence, you are guaranteed to get the same sequence if you set the same seed!

Of course, you might also want to the program to run differently on each occasion. In such cases, you can use the command `time` to set the seed, i.e. write

```
srand(time());
```

The `time` command returns the current time in seconds since some midnight of January 1, 1970, or some such moment. Clearly, you can expect it to be different on each run.

8.6.1 randuv command in simplecpp

In simplecpp we have provided the command randuv which takes two double arguments u,v and returns a random double in the range u through v. Our command calls the C++ supplied function rand, and returns the following value:

```
u + (v-u)*rand()/(1.0 + RAND_MAX)
```

As you can see this value will be between u and v and uniformly distributed to the extent rand is uniformly distributed.

If you want random numbers between integers i,j, you must call randuv(i,j+1) and convert it to an integer. This will give you uniformly distributed integers between i and j.

You can use srand to set the seed as before.

8.7 Concluding remarks

You may find all the suggestions in this chapter to be very cautious, if not paranoid. But when it comes to serious programming, it is better in the long run to be humble and paranoid.

8.8 Exercises

1. Here is a “clever” observation about the digit counting problem. Suppose a number n has d digits. Then $\lfloor n/10 \rfloor$ has $d-1$ digits. Thus we simply count the number of times we can divide by 10 till we get zero and that will be the number of digits of the number. So the program is:

   ```
   main_program{
     int n, d=0;
     cin >> n;

     while(n>0){
       n = n/10;
       ++d;
     }

     cout << "There are "<<d<<" digits.\n";
   }
   ```

   Is this program correct? Would you have written this program if you had followed the process suggested in this chapter? For what values of the input would you test the program?

2. Write the program that finds the smallest circle covering a given set of points. Allow the user to supply the points by clicking on the screen, and show the smallest circle also on the screen. Use Theorem **??** and consider all possible candidates. Later we will see ways by which we can rule out some candidates without checking them!

3. What are good test cases for the smallest covering circle problem?

4. Consider the problem of finding the smallest circle that covers a given set of points. Can you prove the insight presented in the text for solving the problem?

Chapter 9

Functions

In the preceding chapters, we have seen programs to do many things, from drawing polygons and miscellaneous pictures to calculating income tax and finding the greatest common divisor (GCD) and finding roots. It is conceivable that we will want to write more and more complex programs in which some of these operations, e.g. finding the GCD, is needed at many places. One possibility is to copy the code for the operation in as many places as is required. This doesnt seem too elegant, and is also error prone. Wouldnt it be nice, if for each frequently used operation you could somehow construct a "command" that could then be used wherever you want in your program? Just as we have a command for computing the square root, or the trigonometric ratios (Section 1.5) can we build a command that will compute the GCD of two numbers when demanded? This can be done, and how to build such commands is the subject of this chapter.

The term *function* is used in C++ to denote what we have so far informally called a command. In some languages the terms *procedure* or *subprogram* are also used. In what follows, we will use the term *function*.

9.1 Defining a function

Suppose that we indeed need to frequently compute the GCD, and so would like to have a function which computes this. It is natural to choose the name `gcd` for this function. It could take two numbers as arguments, and return their GCD, which could then be used. As an example, suppose you wanted to find the GCD of 24 and 36, and also the GCD of 99 and 47. If we had a `gcd` function as described, then we could write a very simple main program as follows.

```
main_program{
  int a=36,b=24,c=99,d=47;
  cout << gcd(a,b) << endl;
  cout << gcd(c,d) << endl;
}
```

Since we dont already have such a `gcd` function, we must define it. We discuss how to do this next.

```
int gcd             // return-type function-name
  (int L, int S)    // parameter list: (parameter-type parameter-name ...)
{                   // beginning of function body
  int Remainder;

  while(true){
    Remainder = L % S;
    if (Remainder == 0) break;
    L = S;
    S = Remainder;
  }
  return S;
}                  // end of function body
```

Figure 9.1: Defining a function to compute GCD, comments indicate the general form

Basically, in the definition, we must specify what needs to happen when the command is encountered during the execution of the main program. In essence, the idea is to have a small program run, sort of in the background, for computing the GCD. This program, which we will refer to as a *subprogram* must be given the inputs, (in the present case, the values of the numbers whose GCD is to be computed), and some mechanism must be established for getting back the result (in the present case the computed GCD) of the computation to the main program. While the sub-program runs, the main program must simply wait.

Figure 9.1 shows the code for defining gcd. The simplest way to use the definition is to place it in the same file as the main program given earlier, before the main program. If you compile and run that file, then it will indeed print 12 and 1, the GCD respectively of 24, 36 and 99, 47, as you expect. The requirement that the function be placed before the main program is similar to the requirement that a variable must be declared before it is used. We can relax this requirement slightly, as will be seen in Section 9.7.5.

In general, a function definition has the form:

```
type-of-return-value function-name (parameter1-type parameter1-name,
  parameter2-type parameter2-name, ...){
  body
}
```

The definition begins with type-of-return-value, which indicates the type of the value returned by the function. In the GCD example, the function computes and evaluates the GCD, which has type int, so our definition (Figure 9.1) mentions this.

Next is function-name, the name of the function being defined. In our example, we chose to call our function gcd, so that is what the code states. Any valid identifier (Section 3.1.3) can be chosen as a function name.

Next is a parenthesised list of the parameters to the function, together with their types. Parameters are simply variables, with an additional function which we see later. In our case, there are two parameters, L,S both of type int.

Finally, comes the code, body, that is used to compute the return value. The body is expected to be a sequence of statements, just as you would expect in any main program. It can contain declarations of variables, conditional statements, looping statements, everything that can be present in a main program. However, there are two additional features. The code in the body can refer to the parameters, as if they are variables. Further, the the body must contain a return statement, which we explain shortly. We note that the body of the function is taken from the program developed in Section 7.4, with slight modification. First, as remarked there, the code works even if initially the variable Large is given a smaller value than the variable Small. So we have changed the misleading names Large, Small to L, S. Second, the body has a return statement instead of printing out the results.

9.1.1 Execution: call by value

Consider our gcd function and main program. While executing the main program, suppose that control arrives at the call gcd(a,b). We describe the general rule that determines what happens, and also mention what happens in our specific case.

1. The arguments to the call are evaluated. In our case it simply means fetching the values of the variables a,b, viz. 36,24. But in general, the arguments could be arbitrary expressions which would have to be evaluated.

2. The execution of the calling subprogram, i.e. the subprogram which contains the call, main_program, in this case, is suspended. The calling program, main_program in our example, will be resumed later. When resumed, the execution will continue from where it was suspended.

3. Preparations are made to start running a subprogram. The subprogram will execute the code given in the body of the function. The subprogram must be given a separate area of memory so that it can have its own variables. It is customary to refer to this area as the *activation frame* of the function call. Immediately, space is allocated in the activation frame for storing the variables corresponding to the parameters of the function.

   Thus in our case, an activation frame is created corresponding to the call gcd(a,b). The gcd function has two parameters, L, S. So variables, L and S will be created in the activation frame.

4. The value of the first argument is copied to the memory associated with the first parameter. The value of the second argument to the second parameter, and so on.

   Thus, in our case, 36 will be copied into the variable L, and 24 into the variable S in the activation frame created for the call gcd(a,b). Figure 9.2(a) shows the state of the memory at this time. We have referred to the memory area used by main_program as its activation frame. This is customary.

5. Now the body of the called function is executed. The body must refer to variables or parameters stored only in the activation frame of the call.[1] and if space needs to be reserved for variables etc., it is done only inside the activation frame of the call.

[1] We will modify this a bit later.

Thus in case of our program, the code may refer to the parameters `L, S`. The code *cannot* refer to variables `a,b,c,d` because they are not in the activation frame of `gcd(a,b)`. When the first statement of the `body` is executed, it causes the creation of the variable `Remainder`. The space for this is allocated in the activation frame of the call. Such variables are said to be *local* to the call.

6. The `body` of the function is executed until the `return` statement is encountered. The expression following `return` is called the `return-expression` and its value is sent back to the calling program. The value sent back is considered to be the value of the call in the calling program.

   In our case the execution of the function progresses in the usual fashion. In the first iteration of the loop, `L, S` have values 36,24. At the end of this iteration, the values become 24,12. The state of the memory at this point is shown in Figure 9.2(b). In the next iteration, `Remainder` becomes 0, and so the `break` statement is executed. Thus the control exits from the loop, and *return* is reached. The `return-expression` is `S` which has value 12. This value is sent back to the calling program.

7. The activation frame created for the call is not needed any longer, and is destroyed, i.e. that area is marked available for general use.

8. The calling program resumes execution from where it had suspended. The returned value is used as the value of the call itself.

   In our case the call was `gcd(a,b)`, and its value is required to be printed. Thus the value returned, 12, will be printed. After this the next `cout` statement will be executed (in which we will encounter the second call to `gcd`). This will cause an activation frame to be created again etc.

In this model of executing function calls, only the values of the arguments are sent from the calling program to the called function. For this reason, this model is often termed as *call by value*. We will see another model later on.

It is worth considering what happens on the second call to `gcd`, i.e. the call `gcd(c,d)` in the code. The same set of actions would repeat. A new activation frame would be created for this call, and very likely it would use the same memory as was used for the activation frame of the previous call, because we marked that memory available for use. The point to be noted is that each call requires some additional memory, but only for the duration of the execution of the call.

9.1.2 Names of parameters and local variables

We have already said that when a function call executes, it can only access the variables (including the parameters) in its activation frame. In particular, the variables in the calling program (in this case `main_program`) cannot be accessed. So it is perfectly fine if variables in the calling program and the called function have the same name! Note further that when the calling program is executing, the activation frame of the called function does not even exist, so there is no question of any confusion.

Activation frame of `main_program`	Activation frame of `gcd(a,b)`
`a` : 36	`L` : 36
`b` : 24	`S` : 24
`c` : 99	
`d` : 47	

(a) After copying arguments.

Activation frame of `main_program`	Activation frame of `gcd(a,b)`
`a` : 36	`L` : 24
`b` : 24	`S` : 12
`c` : 99	`Remainder` : 12
`d` : 47	

(a) At the end of the first iteration of the loop in `gcd`.

Figure 9.2: Some snapshots from the execution of `gcd(a,b)`

9.2 Nested function calls: LCM

Suppose now that you wish to develop a program to compute the least common multiple (LCM) of two numbers. This is easily done using the following relationship between the LCM, L, and the GCD, G of two numbers m, n:

$$L = \frac{m \times n}{G}$$

It would of course be nice to write a function for the LCM, so that we could invoke it whenever needed, rather than having to copy the code. We could use the above relationship, but that would require us to compute the GCD itself. Does it mean that we need to rewrite the code for computing the GCD inside the function to compute LCM? Not at all. We can simply call the `gcd` function, since we have already written it! So here is how we can define the a function to compute the LCM.

```
int lcm(int m, int n){
  return m*n/gcd(m,n);
}
```

The execution of `lcm` follows the same idea as in our discussion earlier for `gcd`. Suppose in `main` we need to calculate the LCM of 36 and 24. So it contains the expression `lcm(36,24)`. When this expression is encountered, we will need to run a subprogram for `lcm`, which involves creating the activation frame for this call. As this subprogram executes, we will encounter the expression `gcd(m,n)` with `m,n` taking the values 36,24. To process this call, we will need to start a subprogram for `gcd`. So at this point, we will have 3 activation frames in memory, one for `main_program`, one for `lcm(36,24)` and another for `gcd(36,24)`. This is perfectly fine! When the subprogram for `gcd(36,24)` finishes, then the result, 12, will be sent back to the subprogram for `lcm(36,24)`. The result 12, will be used as the value of the call gcd(m,n). Thus the expression m*n/gcd(m,n) can now be evaluated to be 36*24/12=72. This will in fact be the value that the subprogram lcm(36,24) returns back to main_program. At this point, the computation of main_program will resume with the received value.

9.3 The contract view of function execution

While it is important to know how a function call executes, while thinking about functions, a different, metaphorical view is useful.

The idea is to think of a function call as giving out a *contract* to get a job done. We think of the main program as an agent doing its work as described in its program. Suddenly, the agent encounters a statement such as lcm(36,24). Rather than doing the work required to compute lcm(36,24) itself, the main program agent engages another agent. This agent is the subprogram for the call lcm(36,24). The main program agent sends the input data to the subprogram agent, and waits for the result to be sent back. This is not unlike engaging a tailor, giving the tailor the cloth and measurements, and waiting for the tailor to send back a shirt.

The similarity extends further. There is nothing to prevent the tailor from further contracting out the work to others. It so happens, that stitching the collar of a shirt is a specialized job, which most tailors would in fact contract out to collar-specialists. Thus it is possible that we may be waiting for the tailor to send us back the shirt, and the tailor might be waiting for the collar specialist to send back a collar. Notice that this is very similar to main_program) waiting for lcm(36,24) which in turn is waiting for gcd(36,24).

9.3.1 Function specification

A key point to be noted from the tailor example above is that when we ask for a shirt to be stitched, we generally do not worry about what the tailor will do. The tailor may do all the work, or subcontract it out further to one or more craftsmen – that is not our concern. We merely focus on the promise that the tailor has made to us – that a shirt will be delivered to us. We dont worry about how the tailor does it, but we merely hold the tailor to deliver us a good shirt (and at the right time and price, as per what has been agreed). If we tried to worry about what our tailor should be doing, and what our accountant should be doing, and what our doctor should be doing, and so on, we would probably go mad!

Likewise, when we call a function in our program, we do not think of how exactly it will get executed. We merely ask: what exactly is being promised in the execution of this function? The promise, is actually both ways, like a contract and is customarily called the *specification* of the function. The specification of gcd could be as follows:

> A call gcd(m,n) returns the greatest common divisor of m,n, where m,n must be positive integers.

You will notice that the specification lays down the responsibilities of both the calling program, and the called program.

1. Responsibilities of the calling program: To supply only positive integers as arguments. Notice that C++ already prevents you from supplying fractional values when you declare the type of `L, S` to be `int`. However, nothing prevents a calling program from supplying negative values or 0. The specification says that the programmer who wrote the function `gcd` makes no guarantees if you supply 0 or negative values. The conditions that the input values are required to satisfy are often called the *pre-conditions* of the function. In addition, the calling program might also have to deal with *post-conditions*, as will be discussed in Section 9.4.

2. Responsibilities of the called program: If the calling program fullfills its responsibilities, and only if the calling program fullfills its responsibilities, does the called program promise to return the greatest common divisor (or do whatever was expected of it). No guarantees are given if the preconditions are not satisfied. Thus in case of `gcd`, if a negative value or zero is supplied: nonsense values will be returned, or the program may never terminate, or terminate with an error.

It is extremely important to clearly write down the specification of a function. You may sometimes avoid doing so, thinking that the specification is obvious. But it may not be so! For example, a more general definition of GCD might allow one of the numbers to be zero, in which case the other number is defined to be the GCD. If this is the definition a user is familiar with, he/she might supply 0 as the value of the second parameter `n`. This will certainly cause the program to terminate because of a division by zero in the very first step of our code. To prevent such misunderstandings, it is best to write down the specifications in full detail.

The natural place to write down the specification is immediately before the function definition. So your function for `gcd` should really look like the following.

```
int gcd(int L, int S)
// Function for computing the greatest common divisor of integers m,n
// PRE-CONDITION: L, S > 0
{
...
}
```

Please get into the habit of writing specifications for all the functions that you write. Note that in the specification it is important to not write *how* the function does what it does, but only *what* the function does, and for what preconditions.

A description of how the function does what it does, often referred to as the description of the *implementation* of the function is also important. But this should be kept separate from the specification. The description of *how* can be in a separate document, or could be written as comments in the body of the code of the function. For example, the following comment might be useful to explain how the `gcd` function works.

```
// note the theorem: if n divides m, then GCD(m,n) = n.
//             If n does not divide m, then GCD(m,n) = GCD(n, m mod n)
```

This comment could be placed at the beginning of the loop.

9.4 Functions that do not return values

Every function (or command) does not need to return a value. You have already seen such functions, e.g. `forward`, which causes the turtle to move forward, but itself does not stand for any value. The command `forward` is predefined for you, but you can also define new functions or commands that do something and do but do not return a value.

For example, you might wish to build a function which draws a polygon with a given number of sides, and having a certain given sidelength. Clearly, it must take two arguments, an integer giving the number sides, and a `float` giving the side length. Suppose we name it `polygon`. The function does not return any value, so we are required to specify the return type in the definition to be `void`. Also, since nothing is being returned, we merely write `return` with no value following it.

```
void polygon(int nsides, float sidelength)
// draws polygon with specified sides and specified sidelength.
// PRE-CONDITION: The pen must be down, and the turtle must be
// positioned at a vertex of the polygon, pointing in the clockwise
// direction along an edge.
// POST-CONDITION: At the end the turtle is in the same position and
// orientation as at the start.  The pen is down
{
  for(int i=0; i<nsides; i++){
    forward(sidelength);
    right(360.0/nsides);
  }
  return;
}
```

Note the precondition: it states where the polygon is drawn in comparison to where the turtle is pointing. Similarly, we should mention where the turtle is at the end, this will be needed in order to know how to draw subsequently. A condition such as this one, which will be true after the execution of the function, is said to be a *post-condition* of the function. A post-condition is also a part of the specification.

9.5 A text drawing program

We would like to develop a program using which it is possible to write on the screen using our turtle. For example, we might want to write “IIT MUMBAI”. How should we organize such a program?

A natural (but not necessarily the best, see the exercises) way of organizing this program is to have a separate function for writing each letter. For example, we will have a function `drawI` for drawing the letter 'I'. Suppose we decide that we will write in a simple manner, so that the letter 'I' is just a line, without the horizontal lines at the top and bottom.

What is the specification of `drawI`? Clearly it must draw the line as needed. But where should the line get drawn? This must be mentioned in the specifications. It is tempting to say that the line will get drawn at the current position of the turtle, in the direction the turtle is pointing. Is this really what we want? Keep in mind that you dont just want to draw one letter, but a sequence of letters. So it is important to *bring the turtle to a convenient position for drawing subsequent letters.* And what is that convenient position?

Suppose we think of each letter as being contained inside a rectangle. It is customary to call this rectangle the *bounding-box* of the letter. Then we will make it a convention that the turtle must be brought to the bottom left corner of the bounding box, and point towards the direction in which the writing is to be done. Where would we like the turtle to be at the end of writing one character so that the next character can be written easily? Clearly, the most convenient final position is pointing away from the right bottom corner, pointing in the direction of writing. We must also clearly state in the precondition whether we expect the pen to be up or down. Also whether the inter-character space is a part of the bounding box or not. If the space is a part of the bounding box, a natural question arises: is it on both sides of the character or only on one side (which?)? We should not only answer these questions, but must also include the answers in the specification.

Based on the above considerations, `drawI` could be defined as follows.

```
void drawI(float ht, float sp){
/*Draws the letter I of height ht, leaving sp/2 units of space on both
  sides.  Bounding box includes space.
  PRECONDITION: the turtle must be at the left bottom of the bounding-box
  in which the character is to be drawn, facing in the direction of
  writing.  Pen must be up.
  POSTCONDITION: The turtle is at the bottom right corner of the
  bounding-box, facing the writing direction, with pen up.  */

  forward(sp/2);
  penDown();
  left(90);
  forward(ht);
  penUp();
  left(180);
  forward(ht);
  left(90);
  forward(sp/2);
  return;
}
```

Functions for other letters are left as exercise for you. So assume that you have written them. Then to write our message, our main program could be as follows.

```
main_program{
  int ht=100, sp=10;
  turtleSim();
  left(90);     // turtle is pointing East at the beginning.
  drawI(ht,sp);
  drawI(ht,sp);
  drawT(ht,sp);
  forward(sp);
  drawM(ht,sp);
  drawU(ht,sp);
  drawM(ht,sp);
  drawB(ht,sp);
  drawA(ht,sp);
  drawI(ht,sp);
  closeTurtleSim();
}
```

A remark is in order. You will see that there are local variables named `ht` and `sp` in the main program, as well as the functions have parameters called `ht` and `sp`. This is acceptable. When the function is being executed, the execution refers only to its activation frame, and hence the variables in the main program are not visible. When the main program is executing, the activation frame of the functions is not even present, so there is no confusion possible.

9.6 The main program is a function!

The main program that we have been writing, `main_program` is in fact a function that we have been defining, just like all the functions we saw in this chapter. The `simplecpp` changes the phrase `main_program` that you write into the phrase `int main()`, so what you specify as the main program is in fact the body of a function called `main` which takes no arguments, and returns an integer. It is a function which gets called by the operating system when you ask for the program to be run. The `main` function has return type `int` because of some historical reasons not worth understanding. You may also wondering why we havent been writing a `return` statement inside `main` if in fact it is supposed to return an `int`. The C++ compiler we have been using, the GNU C++ compiler, ignores this transgression, that's why!

9.7 Organizing functions into files

It is possible to place the main program and the other functions in different files if we wish. If a program is very large, breaking it up into multiple files makes it easier to manage. If a program is being developed cooperatively by several programmers, then it is natural to ask each programmer to develop different functions, and it would be very inconvenient to have all the functions in the same file as they are being developed. A program can be partitioned into a collection of files provided the following rules are obeyed.

> **Rule 1:** If a certain function `f` is being called by the code in file `F`, then the function `f` must be *declared* inside `F`, textually before the call to `f`. Note that a function definition is a declaration, but not vice versa. We will see what a declaration is shortly.
>
> **Rule 2:** Every function that is called must be present in some file in the collection.

9.7.1 Function Declaration

A function declaration is merely its definition without the body. Here for example are the declarations of `lcm` and `gcd`.

```
int lcm(int m, int n);
int gcd(int m, int n);
```

The names of the parameters can be omitted from declarations, e.g. you may write just `int lcm(int,int);` in the declaration.

Suppose a compiler is processing a file containing the statement `cout << lcm(24,36);`. When it reaches this statement, it needs to be sure that `lcm` is indeed a function, and not some typing mistake. It also needs to know the type of the value returned by `lcm` – depending upon the type the printing will happen differently. Both these needs are met by the a declaration. A declaration of a function `f` provides (a) an assurance that `f` as used later in the program is indeed a function, and that it may not have been defined so far, but it will be defined later in this file itself or in some other file, (b) a description of the types of the parameters to the function and the type of the value returned by the function. Given the declaration, the file can be compiled *except* for the code for executing the function itself, which can be supplied later (Section 9.7.2). Notice that a function definition also provides the information in (a), (b) mentioned above, and hence is a also considered to be a declaration.

As an example, suppose that we have a main program that calls our function `lcm` to find the LCM of 24 and 36. Thus there are 3 functions in our program overall: the function `main`, the function `lcm` and the function `gcd`. Figure 9.3 shows how we could form 3 files for the program.

First consider the file `gcd.cpp`, which contains the function `gcd`. It does not call any other function, and so does not need to have any additional declaration. Next consider the file `lcm.cpp`. This contains the function `lcm` which contains a call to `gcd`. So this file has a declaration of `gcd` at the very beginning. Finally the file `main.cpp` contains the main program. This calls the function `lcm`, so it contains a declaration of `lcm` at the beginning. Note that the main program uses the identifier `cout` to write to the console. For this it needs to include `<simplecpp>`, which says what to do with `cout`. The other files do not contain anything which needs services from `<simplecpp>`, so those do not have the line `#include <simplecpp>` at the top.

There are various ways in which we can compile this program. The simplest possibility is to issue the command

```
s++ main.cpp lcm.cpp gcd.cpp
```

This will produce an executable file which will indeed find the LCM of 24,36 when run.

9.7.2 Separate compilation and object modules

But there are other ways of compiling as well. We can separately compile each file. Since each file does not contain the complete program by itself, an executable file cannot be produced. What the compiler will produce is called an *object module*, and this can be produced by issuing the command

```
//-----------------------------------------------------------------
int gcd(int m, int n){       // return-type function-name
                             //    (argument-type argument-name..){
  int mdash,ndash;           // body begins

  while(m % n != 0){
    mdash = n;
    ndash = m % n;
    m = mdash;
    n = ndash;
  }
  return n;                 // body ends
}
//-----------------------------------------------------------------
```

(a) The file `gcd.cpp`

```
//-----------------------------------------------------------------
int gcd(int, int);         // declaration of function gcd.

int lcm(int m, int n){
  return m*n/gcd(m,n);
}
//-----------------------------------------------------------------
```

(b) The file `lcm.cpp`

```
//-----------------------------------------------------------------
#include <simplecpp>
int lcm(int m, int n);    // declaration of function lcm.

main\_program{
  cout << lcm(36,24) << endl;
}
//-----------------------------------------------------------------
```

(c) The file `main.cpp`

Figure 9.3: The files in the program to find LCM

```
s++ -c filename
```

The option `-c` tells the compiler to produce an object module and not an executable. Here `filename` should be the name of a file, say `main.cpp`. In this case, an object module of name `main.o` is produced. If different programmers are working on different files, they can compile their files separately giving the `-c` option, and they will at least know if there are compilation errors.

We can form the executable file from the object modules by issuing the command `s++` followed by the names of the object modules. Thus for our example we could write:

```
s++ main.o gcd.o lcm.o
```

This use of `s++` is said to *link* the object modules together. The linking process will check that every function that was declared but not defined in some module is defined in some other module. After this check, the code in the different modules is stitched up to form the executable file.

It is acceptable to mix `.cpp` files and `.o` files as arguments to `s++`, e.g. we could have issued the command

```
s++ main.cpp gcd.o lcm.o
```

This would compile `main.cpp` and then link it with the other files. The result `main.o` of compiling will generally not be seen, because the compiler will delete it after it is used for producing the executable.

9.7.3 Header files

Suppose programmers M, G, L respectively develop the functions `main, gcd, lcm`. Then G has to tell L how to declare the function `gcd` in the file `lcm.cpp`. The most natural way of conveying this information is to write it down in a so called `header file`. A header file has a suffix `.h`. A simple strategy is to have a header file `F.h` for every program file `F.cpp` which contains functions used in other files. The file `F.h` merely contains the declarations of all the functions in `F.cpp` that are useful to other files. Thus we might have files `gcd.h` containing just the line `int gcd(int,int)`, and `lcm.h` containing the line `int lcm(int,int)`. Now the programmer L writing the function `lcm` can read the file `gcd.h` and put that line into `lcm.cpp`. However, it is less errorprone and hence more customary that `M` will merely place the inclusion directive

```
#include "lcm.h"
```

in his file instead of the declaration. This directive causes the contents of the mentioned file, (`lcm.h` in this case) to be placed at the position where the inclusion directive appears. The mentioned file must be present in the current directory (or a path could be given). Thus all that is needed in addition is to place the file `lcm.h` also in the directory containing `main.cpp`. Likewise M will place the line `#include "lcm.h"` in `main.cpp`, as a result of which the declaration for `lcm` would get inserted into the file `main.cpp` as needed.

Note that we could have used a single header file, say `gcdlcm.h` containing both declarations.

```
int gcd(int,int);
int lcm(int,int);
```

We could include this single file in main.cpp and lcm.cpp. This will cause both declarations to be inserted into each file, while only one is needed. Having extra declarations is acceptable.

9.7.4 Packaging software

The above discussion shows how you could develop functions and supply them to others. You create a F.cpp file and the F.h file containing declarations of the functions defined in F.cpp. Next you compile the F.cpp file giving the -c option. Then you supply the resulting F.o file and the F.h file to whoever wants to use your functions. They must place the file F.h in the directory containing their source files (i.e. files containing their C++ programs), and place the line #include ‘‘F.h’’ in the files which need the functions declared in F.h. Next, they must also place the file F.o in the same directory and mention it while compiling. Note that they do not need to see your source file F.cpp if you dont wish to show it to them.

9.7.5 Forward declaration

Let us go back to the case in which all functions are in a single file. We suggested in Section 9.1 that a function must be defined before its use (i.e. the call to it) in the file. However, as we discussed in the beginning of Section 9.7, it suffices to have a declaration before the use. So if we wish to put the function definition later, we must additionally place a declaration earlier.

When writing a program with several functions in a single file, many people like to place the main program first, perhaps because it gives a good idea of what the program is all about. We can do this; it is fine to organize the contents of your file in the following order.

```
declarations of functions in any order.
definition of main
definitions of other functions in any order.
```

Of course, other orders are also possible.

9.8 Function size and readability

We began this chapter by proposing functions as a mechanism for avoiding code duplication. This is indeed a very important use of functions. However, functions can also be used to make your code easier to understand to other programmers. Ease of understanding is very important especially when programmers work in teams.

One way to improve understandability of anything is to present it in small chunks. When you write a book it is useful to break it up into chapters. A chapter forms an organizational unit of a book. In a similar manner, a function forms an organizational unit of a program. There are a few thumb rules for breaking long text into chapters or sections. An example of a thumb rule: every idea that is important should have its own chapter, or its own section. There are similar thumbrules for splitting large programs into functions.

When it comes to programming, it is often believed that no function, including main should be longer than one screenful. Even with large displays, this gives us a limit of perhaps 40-50 lines on the length of a function. Basically, you should be able to see the entire logic of the function at a glance: that way it is easier to understand what depends upon what, or spot mistakes. How do we break a program into smaller pieces? So far you have not had the occasion to write programs longer than 40 lines, so this discussion is perhaps a bit difficult to appreciate. You will see later, however, that most programs can be thought of as working in phases. Then you should consider writing each phase as a separate function, and give it a name that describes what it does. These functions could be placed in the same file as the main program, but you will find that this will make the program easier to understand. Another idea is to make a function out any modestly complicated operation you may need to perform. As an example of this, consider the apparently simple action of reading in a value from the keyboard. As noted in Section 6.4, a user might type in an invalid value, or the value may not stand for itself but in fact might indicate that the stream of values has finished. One way is to place the logic for dealing with all this in a function that is called by the main program. This idea is partly explored in the read_marks_into function of Section 11.2.

9.9 Concluding remarks

We have remarked about how a function and the main program can have variables with the same name. The general rule for this, and the notion of the *scope of a variable*, is discussed in Appendix B.

Exercises

1. Write a function that prints the GCD of two numbers.

2. Modify the function polygon so that it returns the perimeter of the polygon drawn (in addition to drawing the polygon).

3. Write a function to find the cube root of a number using Newton's method. Accept the number of iterations as an argument.

4. Write a function to determine if a number is prime. It should return true or false, i.e. be of type bool.

5. Suppose the LCM computation program of Figure 9.3 has been written using a single file, and it is noticed that only the function lcm has been declared and also defined, all other functions are defined but not declared. Show how the program could appear in the file.

Chapter 10

Recursive Functions

We are now in a position to describe to you what is perhaps the most powerful, most widespread problem solving technique ever: recursion. What we are going to present will not really contain any new C++ statements. Rather, what you have learned so far will be used, possibly in a manner which might surprise you, to solve some difficult computational problems in a very compact manner.

A fundamental idea in designing algorithms is *problem reduction*. The notion is very common in Mathematics, where we might say "Using the substitution $y = x^2 + x$ the quartic (fourth degree) equation $(x^2 + x + 5)(x^2 + x + 9) + 7 = 0$ reduces to the quadratic $(y + 5)(y + 9) + 7 = 0$.". Of course, reducing one problem into another is useful only if the new problem is in some sense easier than the original. This is true in our example: quadratic equations are easier to solve than quartic. The strategy of reducing a problem to another is easily expressed in programs: the function we write for solving the first problem will call the function for solving the second problem. We saw examples of this in the previous chapter.

An interesting case of problem reduction is when the new problem is of the *same type* as the original problem. In this case the reduction is said to be *recursive*. This idea might perhaps appear to be strange, but it is in fact very common. Consider the following rules for differentiation:

$$\frac{d}{dx}(u+v) = \frac{d}{dx}u + \frac{d}{dx}v$$

$$\frac{d}{dx}(uv) = v\frac{d}{dx}u + u\frac{d}{dx}v$$

The first rule, for example, states that the problem of differentiating the expression $u + v$ is the same as that of first differentiating u and v separately, and taking their sum. You have probably used these rules without realizing that they are recursive. There are two reasons why these rules work:

1. The reduced problems are actually simpler in some precise sense. In our example, the problem of differentiating u or of differentiating v are indeed simpler than the problem of differentiating $u + v$, because u and v are both smaller expressions than $u + v$. Notice that when we reduce one problem to a problem of another type (non-recursive reduction), the new problem is required to be of a simpler type. For recursive reduction, it is enough if the new problem is of a smaller size.

2. Eventually we must have a way to solve some problems directly – we cannot just keep reducing problems indefinitely. The problems which we expect to solve directly are called the *base cases.* Considering differentiation again, suppose we wish to compute:

   $$\frac{d}{dx}(x \sin x + x)$$

   Then using the first rule we would ask to compute

   $$\frac{d}{dx}x \sin x + \frac{d}{dx}x$$

   Now, the computation of $\frac{d}{dx}x$ is not done by further reduction, i.e. this is a base case for the procedure. So in this case we directly write that $\frac{d}{dx}x = 1$. To compute $\frac{d}{dx}x \sin x$ we could use the product rule given above, and we would need to know the base case $\frac{d}{dx}\sin x = \cos x$.

Even on a computer, recursion turns out to be extremely useful. In this chapter we will see several examples of the idea.

10.1 Euclid's algorithm for GCD

Euclid's algorithm for GCD is naturally expressed recursively, as it turns out. Here is Euclid's theorem, restated for convenience.

Theorem 2 (Euclid) *Let m, n be positive integers. If $m\%n = 0$ then $GCD(m, n) = n$. If $m\%n \neq 0$ then $GCD(m, n) = GCD(n, m\%n)$.*

The theorem essentially says that either the GCD of m, n is n, or it is the GCD of $n, m\%n$. But this is exactly like saying that the derivative of an expression can be written down directly or it is the derivative of some simpler expression. Following the analogy, it would seem tempting, to call `gcd` from inside of itself.

```
int gcd(int m, int n)
// finds GCD(m,n) for positive integers m,n
{
  if(m % n == 0) return n;
  else return gcd(n,m % n);
}
```

And the most interesting thing is that it works! In the last chapter we sketched out the mechanism used to execute functions, and it turns out that the same mechanism will correctly compute the GCD using the above code. We will see an example and a general proof shortly.

The function `gcd` as defined above calls itself. Such functions are said to be recursive. Such functions not only work, but they embody an interesting strategy for solving problems.

Function call	main	gcd(36,24)	gcd(24,12)
Activation Frame contents		m : 36 n : 24	m : 24 n : 12
State of call	Suspended	Suspended	Executing

Code with ▷ showing next statement to be executed	cout <<
	▷ gcd(36,24)
	<< endl;
		if(m % n == 0)
		return n;
		else return
		▷ gcd(n, m % n);
			▷ if(m % n == 0)
			return n;
			else return
			gcd(n, m % n);

Figure 10.1: A snapshot of the execution of recursive `gcd`

10.1.1 Execution of recursive `gcd`

Suppose for the moment that our main program in addition the `gcd` definition above is:

```
int main(){ cout << gcd(36,24) << endl;}
```

Suppose the main program begins execution. Immediately it comes upon the call `gcd(36,24)`. As we know, this causes an activation record to be constructed for the call, and in this activation record the parameters `m,n` are assigned the value 36,24 respectively.

Now the execution begins in the new activation record. The first check, `m % n == 0` fails, because `36 % 24` is 12 and not 0. Thus we execute the `else` part. But this contains the call `gcd(n,m % n)`, i.e. `gcd(24,12)`. Our function call execution mechanism must be used again. Thus another activation record is created, this time for `gcd(24,12)`, and `m,n` in this record are set to 24,12 respectively. Also, the execution of the current call, `gcd(36,24)`, suspends. Figure 10.1 shows the state of the world at this time in the execution.

The execution begins in the new activation record. We execute the first statement which requires us to check if `m % n == 0`, i.e. `24 % 12 == 0`. This is indeed true. So we execute the statement `return n`. This causes 12 to be returned. Where does this value go? It goes back to the place from where the current recursive call was made. Since the current call was made while processing the second activation record, the value 12 is returned there. The second activation then continues its execution. However, there isnt much more left to in in this activation. This code was to return the value of `gcd(24,12)` – now that this value is known, 12, it is returned back. So the value 12 is returned also from the second activation. This goes back to the first activation. The first activation resumes from where it was suspended. As per its code, it prints out the received value, 12, and then `main` terminates.

So as you can see, the correct value was computed and printed.

You might also have observed that the values assigned to `m,n` in successive activation records in this program were in fact the same values that `m,n` received at successive iterations our original non-recursive program in Section 7.4.

10.1.2 Interpretation of recursive programs

In some ways there is nothing more to be said about recursive programs – the last section said it all. We mentioned in the previous chapter that a function call should be thought of as contracting an agent to do the work you want, while you wait (are suspended). If the work taken up by the agent is too complicated, then it is possible that the agent might further sub-contract it out to another (sub-)agent. When this happens, you are waiting for the agent to finish, the agent is herself waiting for the sub-agent to finish, and possibly the chain might be very long. But so what?

Of course, the natural intuition is that you contract out work that you cannot do yourself. So it makes sense for the function `lcm` to contract out the work of `gcd` as was done in the last chapter. But whoever heard of contracting out work that you yourself can do? That is in fact what seems to be happening: the recursive function `gcd` clearly should know how to compute the GCD, and yet it seems to be subcontracting out work!

Suppose you have the task of building a Russian doll, which is a children's toy which looks like a doll, but you can open up the doll to see that there is a doll inside, which contains another doll and so on till some fairly small doll is reached, which cannot be further opened up. Suppose further that we define a k level doll to be a doll which contains $k-1$ dolls inside. So let us say that your task is of building a k level doll. How would you do it recursively?

You would contract it to some craftsman. Imagine that the craftsman builds the outer doll, but does not work on the inner dolls. Instead the task of building the inner $k-1$ dolls, which are really a $k-1$ level doll is subcontracted to another craftsman. And so it goes. This continues until a craftsman is asked to build a 0 level doll, which is just an ordinary doll. This is not subcontracted but built directly. So this doll is sent back to the previous craftsman who adds a doll and makes it a level 1 doll and sends it back, and so on, until you eventually receive the k level doll that you ordered!

This is clearly a strange way of building dolls – but what you should understand for now is that it can work in principle. To prove that the process works correctly, you would use induction. First establish that some craftsman can build a level 0 doll without further subcontracting, the so called *base case.* Next, you must prove that a craftsman can put together a level $i+1$ doll given a level i doll. This is the inductive step.

We see how this works for GCD next.

10.1.3 Correctness of recursive programs

The key to proving the correctness of recursive programs is to use mathematical induction. We first need to have a notion of the *size of the problem being solved.* The induction hypothesis typically states that the program correctly solves problems of a certain size.

As an example we will see how to argue that our code will correctly compute the GCD. The argument is really very similar to that in Section 7.4, but we will state it more directly this time.

In our case, it is natural to choose the value of the second argument as the size of the problem. Our induction hypothesis $IH(j)$ is: `gcd(`i,j`)` correctly computes the GCD of i,j for all i.

The base case is $j=1$. But in the first step of `gcd(`$i,1$`)` we will discover that $i\%1$ is zero, and will report 1 as the answer. This is clearly correct.

So let us assume $IH(1), IH(2), \ldots, IH(j)$. Using these we will prove $IH(j+1)$. So consider the call `gcd(`$i,j+1$`)`. In the first step, we check whether $i\%j+1$ equals 0. This is true if $j+1$ divides i, and in that case $j+1$ is the GCD, which is correctly returned. Suppose then that the condition is false. In that case the algorithm tries to compute and

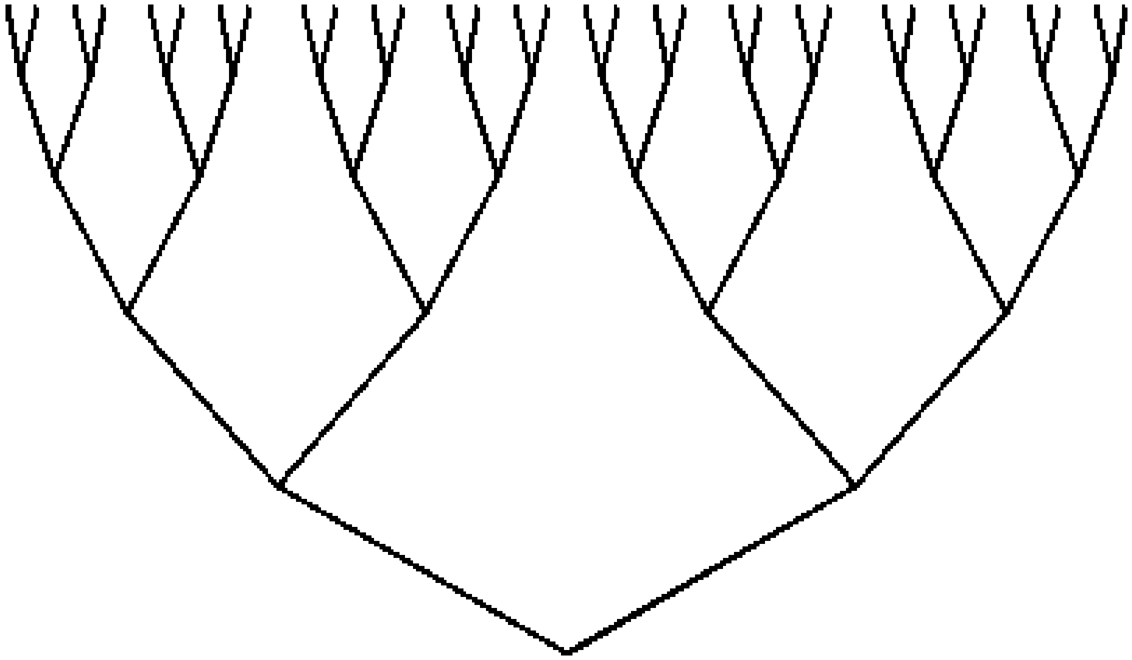


Figure 10.2: An exotic tree

return `gcd(`$j, i\%j+1$`)`, But the second argument in this is the remainder when something is divided by $j+1$, so clearly it must be at most j. Thus by one of our assumptions $IH(1), \ldots, IH(j)$, we know that this call will return correctly, i.e. return $GCD(j, i\%j+1)$. But by Euclid's theorem, we know that this is also $GCD(i, j+1)$, i.e. it is the correct answer. Thus correctness follows by the principle of (strong) induction.

10.2 Recursive pictures

Figure 10.2 shows a picture which we might consider, using some imagination, to be of a tree, say from the African Savannas. This picture has some interesting symmetries. First of course there is a symmetry of reflection about a vertical line through the middle. But also to be noted is the another kind of symmetry: parts of the tree are similar to the whole. The portion of the tree on top of the left branch from the bottom, or the portion on top of the right branch, can each be seen as a tree. In fact we might describe a tree as two small trees on top of a "V" shape formed by the branches at the bottom.

Clearly, the structure is recursive, and so we will write a recursive program to draw such trees. By the way, our interest in trees goes beyond botany; tree diagrams are used in many places. It is customary, however, in many such diagrams, to draw the tree inverted, i.e. growing downward. Thus, a tree might depict the heirarchical structure of many organization, e.g. the root might represent the president, and that may be connected to the vice presidents who report to the president, those in turn to the managers who report to the vice presidents. As you will see later, the manner in which functions are called also have a tree structure. So understanding tree structures and being able to draw them is useful.

It is customary to define the *height* of the tree to be the maximum number of branches you travel over as you move up from the root towards the top along any path. Our tree of Figure 10.2 has height 5. Clearly, we can say that a tree of height h is made up of a *root* with two branches going out, on top of each of which sits a height $h-1$ tree. Of course, to draw the picture, we need more information, for example, what is the length of the branches, and what are the angles at which the branches emerge. For the tree shown, the branch lengths shrink as we go upwards, and so do the angles. Suppose we declare that we want both the branch lengths and the angles between emerging branches to both shrink by a fixed *shrinkage* factor as we go up. Now, if we are given the length of the bottom most branches, and the height of the tree, we should be able to draw the picture.

The code follows the basic observation: to draw a tree of height h, we must draw the root and immediate branches, and two trees of height $h-1$ on top. A tree of height h is just a point, and so nothing need be drawn. As in any drawing program, we must carefully write the pre and post conditions for the turtle. So we will require that at the beginning the turtle to be at the root, pointing upwards (pre condition). After the drawing is finished, we will ensure that the turtle is again at the root and pointing upwards (post condition). Once we fix these pre and post conditions, the program writes itself: we merely have to ensure that we maintain the conditions. Here is what the program looks like.

```
void tree(int height, float length, float angle, float shrinkage)
  // precondition: turtle is at root, pointing up.
  // post condition: same
{
  if(height == 0) return;
                        // 1.  draw the left branch
  left(angle/2);
  forward(length);
                       // 2.  draw the left (sub)tree.
  right(angle/2);
  tree(height-1, length*shrinkage, angle*shrinkage, shrinkage);

                       // 3.  go back to the root
  left(angle/2);
  forward(-length);
                       // 4.  draw the right branch
  right(angle);
  forward(length);
                       // 5.  draw right (sub)tree.
  left(angle/2);
  tree(height-1,length*shrinkage, angle*shrinkage, shrinkage);
                       // 6.  go back to root
  right(angle/2);
  forward(-length);
                       // 7.  ensure post condition
  left(angle/2);
}
```

Clearly, if `height` is 0, we draw nothing and return. Otherwise, the figure is drawn in a series of 7 steps, numbered in the code and explained below.

1. Draw the left branch. For this, the precondition ensures that the turtle is pointing upwards, so it must turn by half the angle that is meant to be between the branches. Then we move forward by the length of the branch.

2. Draw the left subtree. We first turn so that the turtle is facing the top direction, because that is a precondition for drawing trees. Then we recurse. We need to call with `height` one less, and the branch length and angle shrunk by the given shrinkage value.

3. Go back to the root. After drawing the left subtree, its post condition guarantees that the turtle will face directly upwards. To get back to the root we must turn and go backwards, which is accomplished by giving a negative argument to the `forward` command.

4. Draw the right branch. Since the turtle is pointing in the direction of the left branch, we must turn it to the right, and then go forward.

5. Draw the right subtree. This is exactly as we did the left.

6. Go back to the root, as we did after drawing the left subtree.

7. Ensure the post condition. Finally, we want to honour the post condition, so we turn the turtle so that it faces directly upward.

To call the function, we must supply the arguments, but also ensure the precondition. Since we know that at the start the turtle is facing right, we must turn it left by 90 degrees so that it points upwards. So our main program could be the following.

```
int main(){
  turtlesim();
  left(90);

  tree(5,120,120,0.68);

  wait(5);
  closeCanvas();
}
```

10.2.1 Trees without using a turtle

We can draw a tree without using a turtle, as we will show next. However, using a turtle has some advantages, which we remark upon at the end.

The basic idea is to use the Line shape from Chapter 4. We draw the branches using this, and then recurse to draw the subtrees. Note that we must now pass the coordinates of the root as well to each call. For variety, we will also draw a tree with a somewhat different layout. Specifically, we will draw a tree such that it occupies a given rectangular box, with the root appearing at the center of its base. If the coordinates of the root are given, then the box is specified by giving its height H_b and width W_b. We will also assume for simplicity that the for a tree of height H the points at which the branches divide are at heights $H_b/H, 2H_b/H, 3H_b/H, \ldots$. Likewise, when a tree has two subtrees, each subtree is accommodated in a box of half the width of the original box. The code can now be written easily.

```
void tree(int height, float H_b, float W_b,
   float rx, float ry) // coordinates of the root.
{
  if(height > 0){
    float LSRx = rx-W_b/4; // x coordinate of root of Left subtree.
    float RSRx = rx+W_b/4; // x coordinate of root of Right subtree.
    float SRy  = ry-H_b/height; // y coordinate of roots of subtrees.


    Line Lbranch(rx, ry, LSRx, SRy); Lbranch.imprint();
    Line Lbranch(rx, ry, RSRx, SRy); Rbranch.imprint();

    tree(height-1, H_b-H_b/height, W_b/2, LSRx, SRy); // Left Subtree.
    tree(height-1, H_b-H_b/height, W_b/2, RSRx, SRy); // Right Subtree.
  }
}
```

This code is more compact, because we dont have to worry about managing the postconditions of the turtle.

However it should be noted that this code is only useful to grow trees vertically. Suppose you want to orient the tree at an angle of 60 degrees to the vertical, then this code is useless. However, the turtle based code can be used, we merely call it after the turtle is oriented at the required angle. This feature appears useful for drawing many real trees, i.e. the subtrees of many trees appear grow at an angle to the vertical. As a result, to draw realistic looking (botanical) trees, it might be more convenient to use the turtle based code. See Exercise 8.

10.2.2 Hilbert space filling curve

Figure 10.3 shows curves C_1, C_2, C_3 and C_4, left to right. The exercises ask you to understand the recursive structure of the curve, i.e. can you obtain C_i by composing some C_{i-1} curves with some connecting lines. This will help to write a recursive function to draw an arbitrary curve C_n. These curves were invented by the mathematician David Hilbert, and are examples of so called *space-filling* curves.

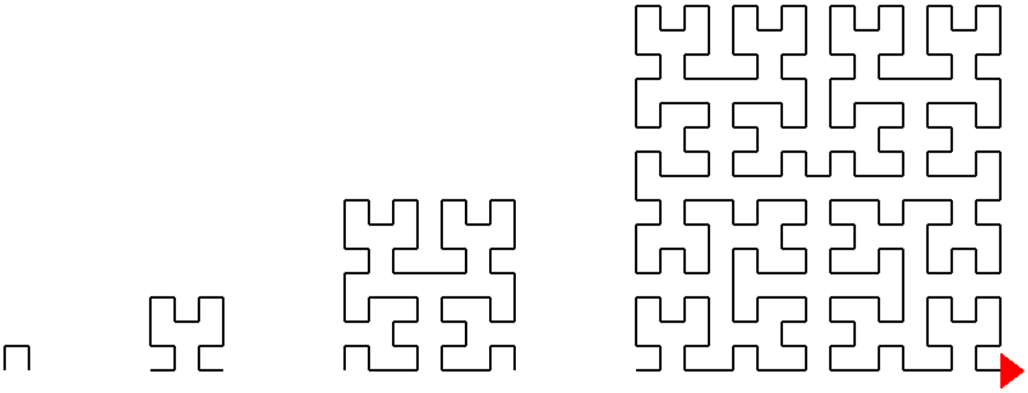


Figure 10.3: Hilbert space filling curves

10.3 Virahanka numbers

Suppose you have an unlimited supply of bricks of heights 1 and 2. You want to construct a tower of height n. In how many ways can you do this? For example, suppose $n = 4$. One possibility is to stack 4 height 1 bricks, i.e. the order of heights considered top to bottom is 1,1,1,1. Other orders are 1,1,2, or 1,2,1 or 2,1,1 or 2,2. You can check by trial and error that no other orders are possible. Thus if you define V_n to be the number of ways in which a tower of height n can be constructed, we have demonstrated that $V_4 = 5$. We would like to write a program that computes V_n given n.

This problem was solved by Virahanka, an Indian prosodist who lived in the 7th century. Virahanka (or quite possibly preceding prosodists, about whom definite information is not known) used the following method to solve the problem, which has been credited to Pingala, who lived around 3rd century BCE. The first observation is that the bottom-most brick is either of height 1 or height 2. You may think this is rather obvious, but from this it follows:

$$V_n = \begin{array}{l}\text{Number of ways of}\\ \text{building a tower of}\\ \text{height } n\end{array} = \begin{array}{l}\text{Number of ways of}\\ \text{building a tower}\\ \text{of height } n \text{ with}\\ \text{bottom-most brick}\\ \text{of height 1}\end{array} + \begin{array}{l}\text{Number of ways of}\\ \text{building a tower}\\ \text{of height } n \text{ with}\\ \text{bottom-most brick}\\ \text{of height 2.}\end{array}$$

Virahanka observed that if you select the bottom-most brick to be of height 1, then the problem of building the rest of the tower is simply the problem of building a height $n - 1$ tower. Thus we have

$$\begin{array}{ccccc} \text{Number of ways of building a tower of height } n \text{ with bottom-most brick of size 1} & = & \text{Number of ways of building a tower of height } n-1 & = & V_{n-1} \end{array}$$

Likewise it also follows that

$$\begin{array}{ccccc} \text{Number of ways of building a tower of height } n \text{ with bottom-most brick of height 2} & = & \text{Number of ways of building a tower of height } n-2 & = & V_{n-2} \end{array}$$

So we have

$$V_n = V_{n-1} + V_{n-2}$$

What we have written above is an example of a *recurrence*, an equation which recursively defines a sequence of numbers, $V_1, V_2, \ldots$ in our case.

Now we are ready to write a recursive program. Clearly, in order to solve the problem of size n, we need a solution to problems of size $n-1$ and $n-2$ respectively. So we have a procedure for reducing the size of the problem, what we need is the base case. Is there a problem that we can solve easily? Clearly, $V_1 = 1$, because a height 1 tower can be built in only 1 way – by using a single height 1 brick.

The trouble is, that this single base case, $V_1 = 1$ is not enough. We should really ask ourselves: will this allow us to find V_2, V_3 and so on? Clearly, we cannot even get V_2 with just this information. However, we can try to find V_2 directly, clearly, there are only 2 ways: 1,1 and 2. So $V_2 = 2$. But now, as we keep recursing, the pairs of numbers we are asking for reduce by one, so eventually the we will want the pair 2,1. But we know the values of V_2 and V_1, and so the recursion will indeed terminate. The program is then immediate.

```
int Virahanka(int n){
  if(n == 1) return 1; // V_1
  if(n == 2) return 2; // V_2
  return Virahanka(n-1) + Virahanka(n-2); // V_{n-1} + V_{n-2}
}
```

If you run this, you will see that it is very slow, even for modestly large n. The reason for it can be seen in Figure 10.4. This figure shows the so called execution tree for the call `Virahanka(6)`.

In an execution tree, the root vertex corresponds to the original call. So we have marked the root in the picture with the argument, 6, to the original call. Out of each vertex we have one downward going edge for every call made. Since `Virahanka(6)` requires `Virahanka` to be called first with argument 5, and then with 4, we have 2 outgoing branches. At the ends of the corresponding branches we have put down 5 and 4 respectively, the arguments for the corresponding calls. This goes on till we get to calls `Virahanka(2)` or `Virahanka(1)`. Since these calls do not make further recursive calls, there are no outgoing branches from the vertices corresponding to these calls.

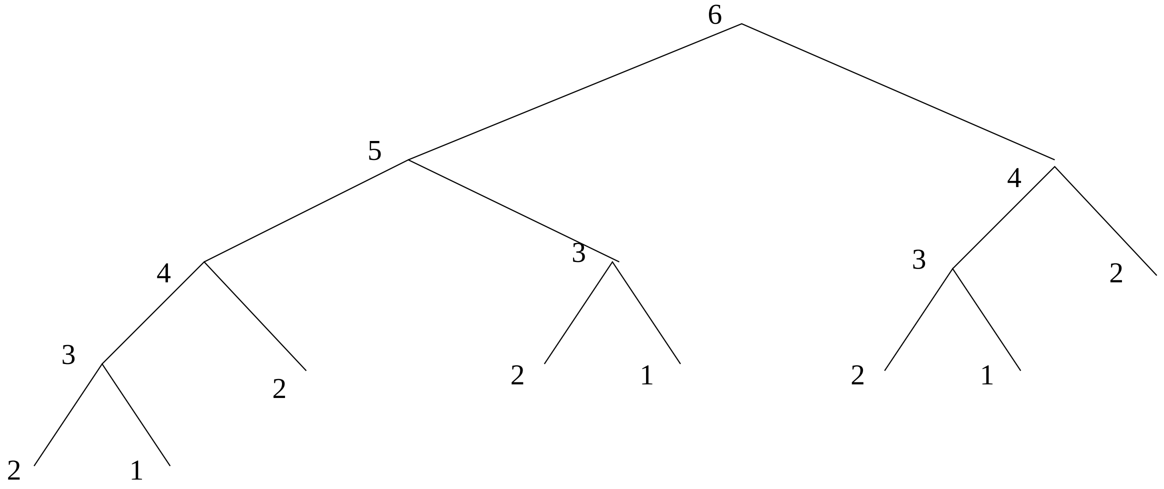


Figure 10.4: Execution tree for `Virahanka(6)`

Note that `Virahanka(4)` is called twice, once as a part of `Virahanka(5)`, and once directly from `Virahanka(6)`. But once we know V_4 using any call to `Virahanka(4)` subsequent calls are not really necessary if we can just remember this value. In fact, you will see that the call `Virahanka(3)` happens 3 times, the call `Virahanka(2)` happens 5 times, and the call `Virahanka(1)` happens 8 times. So the program is quite wasteful. In general, for large n, you can argue (Exercise 4) that while computing V_n, our function will make at least $2^{\lfloor n/2 \rfloor}$ calls to `Virahanka`. Thus if you want to compute V_n by using the recursive algorithm you are expecting to spend time proportional to at least $2^{n/2}$. This is a huge number, and indeed computing something like say V_{45} using the call `Virahanka(45)` takes an enormous amount of time on most computers.

10.3.1 Using a loop

But there is a better way, as you might remember, from Section 6.2.1. The program given there computes Virahanka numbers in order, stopping when the required number is reached. To compute V_n, this program will require $n - 2$ iterations. Each iteration takes a fixed amount of time independent of n. Thus we can say that the total time is approximately proportional to n.

Indeed, you will see that V_{45} gets computed essentially instantaneously using the program of Section 6.2.1.

10.3.2 Historical Remarks

Virahanka actually considered a different problem. He was considering different ways of constructing *poetic meters*, built using syllables of length 1 and length 2. The length of a poetic meter is simply the sum of the lengths of the syllables in the meter. The question he asked was: how many different poetic meters can you compose of a given length? Mathematically, it is the same problem as we solved.

This sequence should look familiar to many readers. Indeed, these numbers are more commonly known as the Fibonacci numbers. But Virahanka is known to have studied them before Fibonacci. In fact it appears that they may have been known in India even before Virahanka. In any case, it seems more appropriate to call these numbers Virahanka numbers rather than Fibonacci numbers.

10.4 The game of Nim

In this we will write a program to play the game of Nim. This game is quite simple, but nevertheless interesting, and our program will contain a key idea which will be useful in all game playing programs.

The game has two players, say White and Black. There are some n piles of stones, the ith pile containing x_i stones at the beginning. We will have different games for different choices of x_i. A move for a player involves the player picking a pile in which there is at least one stone, and removing one or more stones from that pile. The players move alternately, say with White moving first. The player that makes the last valid move, i.e. after which no stones are left, is said to be the winner. Or you may say that the player who is unable to make a move on his turn is the loser.

Here is a simple example. Suppose we have only 2 piles initially with 5 and 3 stones respectively. Suppose White picks 4 stones from pile 1. Then the first pile has 1 stone left and the second has 3. Now Black can win by picking 2 from the second pile: this will leave 1 stone in each pile, and then White can pick only 1 of them, leaving the last one for Black. Of course, White need not have picked 4 stones in the very first move. Is there a different choice for which he can ensure a win? We will leave it to you to observe that White can in fact win this game by picking only 2 stones from the first pile in his first move.

So here is the central question of this section: Given the game position, i.e. number of stones in each pile, determine whether the player whose turn it is to play can win, no matter what the other player plays. In case the position is winning, we would also like to determine a winning move. Note by the way, that when we say winning/losing position, we mean winning/losing for the player whose turn it is to move. It is customary to call the player on move N (next player), and the player not on move P (previous player).

The notion of a winning position is of course intuitively clear to us having played different kinds of games from childhood, including games such as chess and even tick-tack-toe. One way to make this notion formal is to first define a notion of a strategy: a strategy is simply a rule which tells me what move to make for every position. Now a game position G is said to be a winning position if there exists a strategy S for N such that if N plays the rest of the game using S he wins no matter how P plays. Similarly G is said to be a losing position (for N as always) if there exists a strategy S' for P such that P wins if he plays the rest of the game using S', no matter how N plays.

Suppose now that G is some game position, and N has moves $m_1, \ldots, m_k$ in G. Suppose that position G_i results if move m_i is made in position G. Suppose you are told whether each G_i is winning or losing. Could you then determine if G is winning or losing? Turns out that this can be done easily. Suppose some G_i is a losing position. Then if N chooses m_i in G, then we get to G_i which is losing. But now P is on move, and hence G_i is losing for P, and hence G must be winning for N! Thus the only way G can be a losing position is if all G_i are winning.

The above characterization gives us a recursive algorithm for determining if a given position G is winning or losing. We determine all moves m_i possible in G, and the positions G_i they lead to. The we determine (recursively!) whether G_i are winning or losing. If we find some G_i that is losing, we declare G to be winning. Otherwise we declare G to be losing. We know that in order for a recursive algorithm to work we must ensure that 2 things:

1. Each subproblem we are required to solve is simpler than the original. In our case this is true in the sense that each G_i must have at least one stone less than G, and is hence simpler.

2. We can argue that eventually we will reach some ("simplest") problems which we can solve directly. Clearly, as we keep removing stones, we must reach the position in which all piles have zero stones. This position is clearly losing; we know this directly.

Thus we can write a program to determine whether a Nim position is winning or losing. We give this program for the case of 3 piles, but you can see that it can be easily extended for a larger number of piles. The function `winning` given below takes a game position and returns `true` if the position is winning, and `false` if the position is losing.

```
bool winning(int x, int y, int z)
// x,y,z = number of stones in the 3 piles.
// returns true if this is a winning position.
{
  if(x==0 && y==0 && z==0) return false; // base case

  for(int i=1; i<=x; ++i)              // Pick i stones from pile 1
    if (!winning(x-i,y,z)) return true; // if a losing next state is found

  for(int i=1; i<=y; ++i)              // Pick i stones from pile 2
    if (!winning(x,y-i,z)) return true; // if a losing next state is found

  for(int i=1; i<=z; ++i)              // Pick i stones from pile 3
    if (!winning(x,y,z-i)) return true; // if a losing next state is found

  return false;                       // if all next states are winning
}
```

The function can be called using a main program such as the one below.

```
int main(){
  int x,y,z;
  cout << ‘‘Give the number of stones in the 3 piles: ‘‘;
  cin >> x >> y >> z;
  if (winning(x,y,z)) cout << ‘‘Wins.’’ << endl;
  else cout << ‘‘Loses.’’<<endl;
}
```

Our function only says whether the given position is winning or losing, it does not say what move to play if it is a winning position. You can easily modify the program to do this, as you are asked in the exercises.

The logic of our function should be clear. In the given position, we can choose to take stones from either the first pile, the second pile, or the third pile. The first loop in the function considers in turn the case in which we remove `i` stones from the first pile. This leaves the new position in which the number of stones is `x-i,y,z`. We recursively check if this is a losing position. If so, the original position, i.e. in which there are `x,y,z` stones respectively, must be a winning position. Thus we return `true` immediately, as per the discussion above. The subsequent loops consider the cases in which we remove stones from the second and third piles. If we find no losing position after checking all moves, then indeed the original position must be losing, and so we return `false` in the last statement of the function.

10.4.1 Remarks

It turns out that there is a more direct way to say whether a given position is winning or losing. This is very clever, involving writing the number of stones in the piles in binary, and performing additions of the bits modulo 2 and so on. We will not cover this, but you should be able to find it discussed on the web.

Our program is nevertheless interesting, because the general structure of the program applies for many 2 player games. Indeed, recursion is an important tool in writing game playing programs.

10.5 Concluding remarks

A number of points need to be noted.

The most important idea in this chapter concerns algorithm design: if you want to solve an instance of a certain problem. Then it helps to assume that you can solve smaller instances somehow, and ask: will the solution of smaller instances help me in solving the larger instance. It is likely that Euclid discovered his GCD algorithm possibly thinking in this manner. Virahanka probably also discovered the solution to his problem thinking in this manner.

The notion of recurrences is also important.

Tree structures are important in computer science. You will see more uses of trees later in the book.

It is quite likely that Virahanka also solved his problem by thinking recursively. However, writing the program recursively is not always the best way. It is often a good idea to think recursively for the purpose of solving problems. But we must remember that sometimes it may not be best to write the program recursively.

Finally, it should be noted that V_n is at least $2^{n/2}$, and hence even for modest value of n it will not fit in an `int`. Use `long long` so that you can work with somewhat larger values. If you use `double` you can work with much larger values of n, but they will be correct only to 15 digits or so.

Exercises

1. Consider an equation $ax + by = c$, where a, b, c are integers, and the unknowns x, y are required to be integers. Such equations are called Diaphontine equations. Devise a method to find a solution to Diaphontine equations. Hint1: reduce the problem to solving $a'x' + b'y' = c$ where a', b' is in some sense smaller than a, b by doing a variable substitution, e.g. $x' = x \pm y, y' = y$. Hint2: Suppose $a = 1$. Show that in this case an integer solution is easily obtained. Write a program which takes a, b, c as input and gives a solution.

2. Deduce the general structure of Hilbert space filling curves by observing Figure 10.3. Write a program to draw a Hilbert space filling curve V_n given n.

3. Write a program that prints the table V_i versus i.

4. Let B_n denote the number of branches in the recursion tree for V_n. Thus $B_6 = 14$, considering Figure 10.4. Note that each branch ends in a call to `Virahanka`, hence B_n gives a good estimate of the number of operations needed to compute V_n. (a) Write a recurrence for B_n and use it to write a program that computes B_n. What are the base cases for this? Make sure your answer matches the branches in the trees of Figure 10.4. (b) Argue using induction that $B_n \geq 2^{\lfloor n/2 \rfloor}$ for $n \geq 3$.

5. Prove using induction that $2^{\lfloor n/2 \rfloor} \leq V_n \leq 2^n$. Based on this what data type would you use for computing V_{80}? If it helps you may note the stonger result $V_n \leq 1.62^n \leq 2^{n/1.43}$.

6. Suppose you call the function `gcd` on consecutive Virahanka numbers V_n, V_{n+1}. There is something interesting about the arguments to the successive recursive calls. What is it? The *depth* of the recursion is defined to be the number of consecutive recursive calls made, each nested inside the preceding one. What is the depth of the recursion for the call `gcd(`V_{n+1}, V_n`)`? Based on this, argue that the time taken by `gcd` when called on n bit numbers could be as large as $n/2$.

7. Consider the `gcd` function developed in the chapter. Suppose the initial call is with arguments m_0, n_0 and successive calls are made with arguments m_1, n_1, then m_2, n_2, then m_3, n_3 and so on. Show that $n_{i+2} \leq n_i$. Based on this argue that a call to `gcd` with n bit numbers will have depth of recursion at most $2n$. In other words, the time to compute the `gcd` will be at most proprortional to the number of bits in the numbers.

8. More commonly, (botanical) trees have a single trunk that rises vertically, and then splits into branches. So you could consider a tree to be "one vertical branch, with two trees growing out of it at an angle". Draw trees expressing this idea as a recursive program. You could consider variations such as: branches grow out of the trunk, which may continue further. Express these tree growth patterns recursively as well. .

9. Write a function `draw_Hem` that draws the recursion tree for calls to `Virahanka`, i.e. `draw_Hem(6)` should be able to construct the tree shown in Figure 10.4.

10. The tree drawn in Figure 10.2 is called a complete binary tree. Binary, because at each branching point there are exactly 2 branches, or at the top, where they are no branches. Complete, because all branches go to the same height. You could have an incomplete binary tree also, say you only have one branch on one side and the entire tree on the other.

    Write a program which takes inputs from the user and draws any binary tree. Suppose to any request the user will only answer 0 (false) or (true). Device a system of questions using which you can determine how to move the turtle. Make sure you ask as few questions as possible.

11. Suppose you want to send a message using the following very simple code. Say your message only consists of the letters 'a' through 'z', and in the code your merely replace the ith letter by i. Thus 'a' will be coded as 1, 'b' as 2, and so on till 'z' by 26. Further, there are no separators between the numbers corresponding to each letter. Thus, the word "bat" will be coded as the string 2120. Clearly, some strings will have more than one interpretation, i.e. the string 2120 could also have come from "ut". Suppose you are given a string of numbers, say 1 digit per line (so 2120 will be given on 4 lines). You are to write a program that takes such a sequence of numbers and prints out the number of ways in which it can be interpreted. You are free to demand that the input be given from the last digit if you wish.

12. There are many variations possible on the game of Nim as described above. One variation is: the player who moves last loses. How will you determine whether a position is winning or losing for this new game?

13. In another variation, you are allowed to pick either any non-zero number of stones from a single pile, or an equal number of stones from two piles. Write a function that says whether a position is winning for this game.

14. Write a function which returns a 0 if the given position is losing, but if the position is winning, returns a value that somehow indicates what move to make. Decide on a suitable encoding to do this. For example, to indicate that s stones should be picked from pile p, return the number $p \times 10^m + s$, where 10^m is a power of 10 larger than x, y, z the number of stones in the piles for the given position. With this encoding, the last m digits will give the number of stones to pick, and the more significant digits will indicate the pile number. Some of you might wonder whether we can return pairs of numbers out of a function (without having to encode them as above) – we will see how to do it later in the book.

15. Write a recursive function for finding x^n where n is an integer. Try to get an algorithm which requires far fewer than the $n-1$ multiplications needed in the natural algorithm of multiplying x with itself. Hint: Show that k multiplications suffice if $n = 2^k$ is a power of 2. Then build up on this.

Chapter 11

More on Functions

In this chapter we discuss a miscellaneous set of advanced features related to functions.

We begin in Section 11.1 by stating some relatively simple things we would like functions to do, but which cannot be done using what we have learnt so far. The way to overcome these difficulties is to use the mechanism of *call by reference* for passing parameters. We study this next. We also study the closely related notion of *pointers*, and see how this can also overcome the said difficulties.

We then discuss the notion of function pointers, using which we can effectively pass one function as argument to another function. We then study how default values can be specified for some of the parameters to a function. Finally, we consider the notion of function *overloading*, i.e. how the same function name can be used to define more than one functions.

11.1 Some difficulties

There are a few seemingly simple things we cannot do using our current notion of a function. For example, we might want to write a function which takes as arguments the Cartesian coordinates of a point and returns the Polar coordinates. This is not immediately possible because a function can only return one value, not two. Another example is: suppose we want to write a function called `swap` which exchanges the values of two integer variables. Suppose we define something like the following.

```
void swap(int a, int b){  // will it work?
  int temp;
  temp = a;
  a = b;
  b = temp;
}
```

If we call this by writing `swap(p,q)` from the main program, we will see it does not change the values of `p,q` in the main program. The reason for this is that when `swap` executes, it does exchange the values `a,b`, but `a,b` are in the activation frame of `swap`, and their exchange does not have any effect on the values of `p,q` which are in the activation frame of the main program.

As a third example, consider the mark averaging program from Chapter 6. An important step in this program is to read the marks from the keyboard and check if the marks equal 200. If the marks equal 200, then the loop needs to terminate. Here is an attractive way to write the program.

```
int main(){
  float nextmark, sum=0;
  int count=0;

  while(read_marks_into(nextmark)){  // will this work?
    sum = sum + nextmark;
    count = count + 1;
  }

  cout << ‘‘The average is: ‘‘ << sum/count << endl;
}
```

Our hope is that we can write a function `read_marks_into` that will behave in the following manner. It will read the next mark into the variable given as the argument, and also return a true or false depending upon whether the reading was successful, i.e. true if the value read was not 200, and false if it was. But what we have learned so far does not allow us to write this function: The value of the argument `nextmark` will be copied to the parameter of the function, but will not be copied back.

It turns out that all the 3 problems listed above have a nice solution in C++. This solution is based on another way of passing arguments to function, called *call by reference*. We will see this next.

Following that we will see how the problem is solved in the C language. As you might know, C++ is considered to be an enhanced version of C. There are a number of reasons for discussing the C solution. First of all, it is good to know the C solution because C is still in use, substantially. Also, you may see our so called C solution in C++ programs written by someone, because essentially all C programs are also C++ programs. Second, the C solution uses the notion of *pointers*, which are needed in C++ also. Finally, the C solution is in fact a less magical version of the call by reference solution of C++. So in case you care, the C solution might help you understand “what goes on behind the scenes” in call by reference.

11.2 Call by reference

The idea of call by reference is simple: when you make a change to a function parameter during execution, you want the change to be reflected in the corresponding argument? Just say so and it is done! The way to “say so” is to declare the parameter whose value you want to be reflected as a *reference parameter*, by adding an & in front of the name of the parameter. So here is how we might write the function to convert from Cartesian to Polar.

```
void Cartesian_To_Polar(float x, float y, float &r, float &theta){
  r = sqrt(x*x + y*y);
  theta = atan2(y,x);
}
```

In this function, `r` and `theta` have been declared to be reference parameters. No storage is allocated for a reference parameter in the activation frame of the function, nor is the value of the corresponding argument copied. Instead, during the execution of the function, a reference parameter directly *refers* to the corresponding argument. Hence whatever changes the function seems to make to a reference parameter are really made to the corresponding argument directly.

This can be called in the normal way, possibly as follows.

```
int main(){
  float x1=1.0, y1=1.0, r1, theta1;
  Cartesian_To_Polar(x1,y1,r1,theta1);
  cout << r1 << ‘‘ ‘‘ << theta1 << endl;
}
```

Here is how the call `CartesianToPolar(x1,y1,r1,theta1)` executes. The values of `x1,y1` are copied to the corresponding parameters `x,y`. However, as mentioned, the values of `r1, theta1` are not copied. Instead, all references to `r, theta` in the function are deemed to be references to the variables `r1,theta1` instead! Thus, as `CartesianToPolar` executes, the assignments in the statements `r=...` and `theta=...` get made to `r1` and `theta1` directly. So indeed, when the function returns, the variable `r1` would contain $\sqrt{1+1} = \sqrt{2} \approx 1.4142$, and `theta1` would contain $\tan^{-1} 1 = \pi/4 \approx 0.785$, and these would be printed out.

The function to swap variable values can also be written in a similar manner.

```
void swap2(int &a, int &b){
  int temp;
  temp = a;
  a = b;
  b = temp;
}
```

This can be called as follows.

```
int main{
  int x=5, y=6;
  swap2(x,y);
  cout << x << " " << y << endl;
}
```

Both the arguments of `swap2` are references, and so nothing is copied into the activation area of `swap2`. The parameters `a,b` refer directly to `x,y`, i.e. effectively we execute

```
temp = x;
x = y;
b = temp;
```

This will clearly exchange the values of `x,y`, so at the end “6 5” will be printed.

Our last example is also easy to write.

```
bool read_marks_into(int &var){
  cin >> var;
  return var != 200;
}
```

This function will work with the main program as given earlier. When the function executes, the first line will read a value into `var`. But `var` is a reference for the corresponding parameter `nextmark`, and hence the value will in fact be read into `nextmark`. The expression `var != 200` is true if `var` is not 200, and false if it is 200. So the while loop in the main program will indeed terminate as soon as 200 is read. Continuing the discussion at the end of Section 6.3, we note that perhaps this is the nicest way of writing the mark averaging program: we do not duplicate any code, and yet the loop termination is by checking a condition at the top, rather than using a break statement in the body.

11.2.1 Remarks

Call by reference is very convenient. However two points should be noted.

The manner by which we specify arguments of a function in a call is the same, no matter whether we use call by value or by reference for a parameter. This makes it hard to read and understand code. When we see a function call, we need to either find the function definition or declaration (Section 9.7.1) to know which of the arguments, if any, correspond to reference parameters, and hence might change when the function returns. The C language solution which uses pointers, discussed next, does not have this drawback. On the other had it has other drawbacks, as you will see.

If a certain parameter in a function is a reference parameter, then the corresponding argument must be a variable. For example, we cannot write `swap2(1,z)`. This would make `a,b` refer to `1,z` respectively and then statements in the function such as `a = b;` would have to mean something like `1 = z;`, which is meaningless. So supplying anything other than a variable is an error if the corresponding parameter is a reference parameter. However, do see Section 14.1.4.

11.2.2 Reference variables

In the discussion above we noted that a reference parameter should be thought of as just a name, what it is a name of is fixed only when we make the call. In a similar manner, we can have reference variables also.

```
int x = 10;
int &r = x;
cout << r << endl;
r = 20;
cout << x << endl;
```

The first statement defines the variable `x` and assigns it the value 10. The second statement declares a reference `r`, hence the `&` before the name. In the declaration itself we are obliged to say what variable `r` is a name of. This is specified after `=`. Thus the second statement declares the integer reference `r` which is another name for the variable `x`. In the third statement, we print `r`, since this is a name for the variable `x`, the value of that, 10, gets printed. In the fourth statement, we assign 20 to `r`, but since `r` is just a name for `x`, it really changes the value of `x`. Finally, this changed value, 20, gets printed in the last statement.

The utility of reference variables will become clear later, in Section 21.3.2.

11.3 Pointers

We first discuss pointers in general, and then say how they are helpful in solving the problems of Section 11.1.

We know from Section 2.4 that memory is organized as a sequence of bytes, and the `i`th byte is supposed to have address `i`, or be at address `i`. When memory is allocated to a variable, it gets a set of consecutive bytes. The address of the first byte given to the the variable is also considered to be the address of the variable.

11.3.1 "Address of" operator `&`

C++ provides the unary operator `&` (read it as "address of") which can be used to find the address of a variable. Yes, this is the same character that we used to mark a parameter as a reference parameter, and there is also a binary operator `&` (Section G). But you will be able to tell all these apart based on the context. Here is a possible use of the unary `&`.

```
int p;
cout << &p;
```

This will print out the address of `p`. Note that the convention in C++ is to print out addresses in hexadecimal, so you will see something that begins witn `0x`, which indicates that following it is an hexadecimal number. Note that in hexadecimal each digit takes value between 0 and 15. Thus some way is needed to denote values 10, 11, 12, 13, 14, 15, and for these the letters `a,b,c,d,e,f` respectively are used.

11.3.2 Pointer variables

We can store addresses into variables if we wish. But for this we need to define variables of an appropriate type. For example, we may write:

```
int p=15;
int *r;                            // not ‘‘int multiplied by r’’!  See below.
r = &p;
cout << &p << " " << r << endl;
```

The first statement declares a variable `p` as usual, of type `int`. The next statement should really be read as `(int*) r;`, i.e. `int*` is the type and `r` is the name of the declared variable. The type `int*` is used for variables which are to be used for storing addresses of `int` variables. This is what the third statement does, it stores the address of the `int` type variable `p` into `r`. If you execute this code, you will see that the last statement will indeed print identical hexadecimal numbers.

Address	Content	Remarks
104 105 106 107	15	Allocated to `p`
108 109 110 111	104	Allocated to `r`

Figure 11.1: Picture after executing `r = &p;`

Figure 11.1 schematically shows a snapshot of memory showing the effect of storing the address of `p` into `r`. In this we have assumed that `p` is allocated bytes 104 through 107, and `r` is allocated bytes 108 through 111. The address of `p`, 104, appears in bytes 108 through 111, as a result of the execution of `r = &p;`.

Likewise we may write:

```
float q;
float* s = &q;
```

Here we have declared and initialized `s` in the same statement. Note that `float*` and `int*` are different types, and you may not write `s = &p;` or `r = &q;`.

Variables `r,s` are said to be `pointers` to `p,q`. In general, variables of type `T*` where `T` is a type are said to be pointer variables.

Finally, even though we use integers to number memory locations, it is never necessary in C++ programs to explicitly store a specific constant, say 104, into a pointer variable. If you somehow come to know that 104 is the address of a certain variable `v`, and so you want 104 stored in some pointer variable `w`, then you can do so by writing `w = &v;`, without using the number 104 itself. In fact, it is a compiler error in C++ to write something such as `w=104`, where `w` is a pointer, e.g. of type `int*`. Because you dont need to write this, if you actually do, it is more likely to be a typing mistake. So the compiler flags it as an error.

Finally it should be noted that C++ declarations are a bit confusing. The following

```
int* p, q;       // both pointers?
```

declares `p` to be a pointer to `int`, while `q` is simply an `int`. Even if you put no space between `int` and `*` in the above statement, the `*` somehow "associates" with `p` than with `int`.

11.3.3 Dereferencing operator *

If we know the address of a variable, we can get back that variable by using the *dereferencing operator*, `*`. Very simply put, the unary `*` can be considered to be the inverse of `&`. The character `*` also denotes the multiplication operator, and is also used in declaration of pointer variables, but it will be clear from the context which operator is meant.

Formally, suppose xyz is of type T* and has value v. Then we consider the memory at address v to be the starting address of a variable of type T, and *xyz denotes this variable. The unary * is to be read as "content of", e.g. an expression such as *xyz above is to be read as "content of xyz".

For an example, consider the definitions of p,r as given above. Then to find what *r means, we note that r is of type int* and has value &p. Thus *r denotes a variable of type int stored at address &p. But p is exactly such a variable. Hence *r denotes the variable p itself. Thus, if *r were to appear on the left hand side of an assignment statement, we would really be storing a value into p. If *r appeared on the right hand side of an assignment, or in an expression, we would be using the value of p in place of the expression *r. Thus we may write (after the code int p=15; int *r; r = & p;):

```
*r = 22;
int m;
m = *r;
```

In the first statement, we would store 22 into p. In the third statement, we would store the value of p, 22 in this case, into m.

11.3.4 Use in functions

We first note that functions can take data of any type as arguments, including types such as int* or float*. Thus we can write a function to compute the polar coordinates given Cartesian as follows.

```
void CartesianToPolar(float x, float y, float* pr, float* ptheta){
  *pr = sqrt(x*x + y*y);
  *ptheta = atan2(y,x);
}
```

This could be called as follows.

```
int main{
  float r,theta;
  CartesianToPolar(1.0, 1.0, &r, &theta);
  cout << r << ‘‘ ‘‘ << theta << endl;
}
```

Let us first make sure that the types of the arguments in the call and the parameters in the function definition match. The first and second parameters, x, y are required to be a float, and indeed the first and second arguments are both 1.0, of type float. The third parameter pr is of type float*. The third argument is the expression &r, which means the address of r. Since r is of type float, the type of &r is indeed float*, and hence the type of the third argument and the third parameter match. Similarly the type of the fourth argument &theta is also seen to match the type float* of the fourth parameter. So clearly our program should compile without errors.

Let us see how this will execute. When the function CartesianToPolar is called, none of the parameters are reference parameters, and so all arguments have to be copied first. So

1.0 is copied to the parameter x in the activation frame of CartesianToPolar. The second argument 1.0 is copied to y. The third argument &r is copied to pr, and finally the fourth argument &theta is copied to ptheta.

Then the body of the function is executed. The first statement is *pr = sqrt(x*x + y*y);. The right hand side evaluates to $\sqrt{2}$, because x and y are both 1. This value is to be placed in the variable denoted by the left hand side. Now *pr is interpreted exactly as described in Section 11.3.3. Given that pr is of type float*, the expression *pr denotes that float variable whose address appears in pr. But we placed the address of r of the main program in pr. Hence *pr denotes the variable r of the main program. Hence the statement *pr=sqrt(x*x + y*y), even if it appears in the code of CartesianToPolar will store $\sqrt{2}$ into the variable r of main.

Next let us consider the statement *ptheta = atan2(y,x);. Since y,x are both 1, the arctangent will be found to be $\pi/4 \approx 0.785$. Reasoning as before, the expression *ptheta will denote the variable theta of the main program. Thus 0.785 will be stored in theta of main. After this the call will terminate. When the execution of main resumes, $\sqrt{2}$ and 0.785 would get printed by the last statement in main.

We next consider the swap function. It should be clear to you now what we should do: instead of using the variables as arguments, we should use their addresses. Here is the function.

```
void swap(int* pa, int* pb){
  int temp;
  temp = *pa;
  *pa = *pb;
  *pb = temp;
}
```

It may be called by a main program as follows.

```
int main{
  int x=5, y=6;
  swap(&x,&y);
  cout << x << ‘‘ ‘‘<< y << endl;
}
```

The arguments to the call are &x, &y, having type int*, because x,y are of type int. Thus they match the types of the parameters of the function. Thus our program will be compiled correctly.

So let us consider the execution. The address of x will be copied into pa, and the address of y into pb. Thus we may note that *pa in swap will really refer to the variable x of the main program, and *pb in swap will refer to the variable y of the main program. The statement temp = *pa; will cause the value of x to be copied to temp. In the next statement, the value of y is copied to x. The last statement causes the value in temp, i.e. the value in x at the beginning to be copied to y (which is what *pb denotes). The function call completes. The main program then resumes and will print the exchanged values, 6 and 5.

The changes required to the main program for mark averaging and the function read_marks_into are left as exercises.

11.3.5 Reference vs. Pointers

You have seen that there are two ways of writing the functions `Cartesian_To_Polar`, `swap` and `read_marks_into`. Which one is better?

Clearly, the functions are easier to write with call by reference. So that is clearly to be recommended in C++ programs.[1]

11.4 Function Pointers

In Section 7.2 we discussed the bisection method for finding the roots of a mathematical function $f(x)$. Ideally, we should have written the bisection method itself as a C++ function, to which you pass the mathematical function $f(x)$ as an argument. This was not how we wrote the method then. Instead, we presented code for the method in which the code for evaluating $f(x)$ (not a call to it) was inserted as needed. But we can do better now.

Figure 11.2 shows how this can be done. This code contains C++ equivalents of two mathematical functions, $f(x) = x^2 - 2$, and $g(x) = \sin(x) - 0.3$. The single function `bisection` is used to find the roots of both these functions. We will explain how `bisection` works shortly, but first consider the main program. As you can see, `main` calls `bisection` for finding each root. Consider the first call. The first two arguments to the call are the left and right endpoints of the interval in which we know the function changes sign. We have used the same endpoint values as `xL,xR` of Section 7.2, viz. 0,2. The next argument is `epsilon`, which gives the acceptable error in the function value. For this we have chosen the value 0.0001, instead of reading it from the keyboard. The last argument somehow supplies the function `f` whose root is to be computed. Exactly why we need to write `&f` will become clear shortly. The second call is similar. We are asking to find a root in the interval 0 to `PI/2`, again tolerating an error of at most 0.0001. The last two lines merely print out the answers and check if the square of the first answer is close to 2, and the sine of the second answer is close to 0.3

First, the key idea. As you know from Section 2.7, code and data are both placed in memory. Just as we can identify a variable by its starting address, we can identify a function also by the address from where its code starts. Thus C++ has the notion of a *function pointer*, which you could think of as the starting address of the code corresponding to the function. Once we have a pointer to a function, we simply dereference it and use it! Thus, the expressions `&f` and `&g` are merely pointers to our functions `f,g`. So all that remains to explain is how the function bisection will dereference and use them.

The name of the last parameter of bisection is `pf`. We explain its complicated looking declaration soon. In the body, this parameter `pf` indeed appears dereferenced. In the first line it appears as `(*pf)(xL)`, where the parentheses are necessary because just writing `*pf(xL)` will be interpreted as `*(pf(xL))` which is not what we want. Noting that dereferencing

[1] Some of you are probably wondering: when a function executes, and some parameter is a reference parameter, how does the computer know what variable the parameter refers to? A simple answer is: at the time of the call, C++ automatically sends the address of the variables referred to by the reference parameters to the function activation frame. Also, during the function execution, C++ itself dereferences the address of the reference variables, and gets to the variables as needed. So in other words, the operations of sending addresses and dereferencing them that had to be manually written out in C are performed "behind the scenes" by C++.

```
float f(float x){
  return x*x -2;
}

float g(float x){
  return sin(x) - 0.3;
}

float bisection(float xL, float xR, float epsilon, float (*pf)(float x))
// precondition: f(xL),f(xR) have different signs. ( >0 and <=0).
{
  bool xL_is_positive = (*pf)(xL) > 0;
  // Invariant: x_L_is_positive gives the sign of f(x_L).
  // Invariant: f(xL),f(xR) have different signs.

  while(xR-xL >= epsilon){
    float xM = (xL+xR)/2;
    bool xM_is_positive = (*pf)(xM) > 0;
    if(xL_is_positive == xM_is_positive)
      xL = xM;  // maintains both invariants
    else
      xR = xM;  // maintains both invariants
  }
  return xL;
}

int main(){
  float y = bisection(1,2,0.0001,&f);
  cout << "Sqrt(2): " << y << " check square: " << y*y << endl;

  float z = bisection(0,PI/2,0.0001,&g);
    cout << "Sin inverse of 0.3: " << z << " check sin: " << sin(z) << endl;
}
```

Figure 11.2: Bisection method as a function

works exactly as we expect, the expression `(*pf)(xL)` will indeed evaluate to `f(xL)` when `pf` has value `&f`. Similarly the expression `(*pf)(xM)` will evaluat `f(xM)`. Thus this code will work exactly like the code in Section 7.2 for the first call.

So the only thing that remains to be explained now is the cryptic declaration of the last parameter of `bisection`. Basically we need a way to say that something is a function pointer. Note however, it does not make sense to pass a function such as `gcd` (which takes 2 `int` arguments and returns an `int`). Pointers to only certain *types* of functions are acceptable as arguments to bisection: specifically pointers to functions that take a single `float` argument and return a `float` as result.

First, let us consider how to declare a function that takes a `float` as argument and returns a `float` result. If the name of the function is `pf`, then we know from Section 9.7.1 that it can be declared as:

```
float pf(float);  // pf is a function taking float and returning float
```

But now we simply note the general strategy for declaring pointers: if a declaration declares name `v` to be of type `T`, then replace `v` by `*v` and the new declaration will declare `v` to be pointer to T. Doing this we get what we wanted.

```
float (*pf)(float);  // *pf is function taking float and returning float
           // pf is pointer to function taking float and returning float
```

where the parentheses have been put to avoid associating `*` with `float`.

Declaring pointers is somewhat tricky and cryptic. It takes a bit of practice to write such declarations and even read them. To take another example, this is how you would declare `ph` to be a pointer to a function which takes a `double` and `int` as argument and returns a `float`.

```
float (*ph)(double,int);
```

Perhaps the best way to read it is the reverse of what we did above. Replace `*ph` by `h` and observe that `h` must be a function taking `double,int` arguments and returning `float`. Hence `*ph` must be a pointer to such a function.

11.4.1 Some simplifications

The C++ standard allows you to drop the operator `&` while passing the function, and also the dereferencing operator `*` while calling the function. Unfortunately, this does not help in the trickiest part, the declaration of a function pointer parameter.

11.5 Default values of parameters

It is possible to assign default values to parameters of a function. If a particular parameter has a default value, then the corresponding argument may be omitted while calling it. The default value is specified by writing it as an assignment to the parameter in the parameter list of the function definition. Here is a `polygon` function in which both parameters have default values.

```
void polygon(int nsides=4, float sidelength=100)
{
  for(int i=0; i<nsides; i++){
    forward(sidelength);
    right(360.0/nsides);
  }
  return;
}
```

Given this definition, we are allowed to call the function either by omitting the last argument, in which case the `sidelength` parameter will have value 100, or by omitting both parameters, in which case the `nsides` parameter will have value 4 and `sidelength` will have value 100. In other words, we can make a call `polygon(5)` which will cause a pentagon to be drawn with side length 100. We can also make a call `polygon()` for which a square of sidelength 100 will be drawn. We are free to supply both arguments as before, so we may call `polygon(8,120)` which will cause an octagon of sidelength 120 to be drawn.

In general, we can assign default values to any suffix of the parameter list, i.e. if we wish to assign a default to the ith parameter, then a default must also be assigned to the $i+1$th, $i+2$th and so on.

Further, while calling we must supply values for all the parameters which do not have a default value, and to a prefix of the parameters which do have default values. In other words, if the first k parameters of a function do not have default values and the rest do, then any call must supply values for the first j parameters, where $j \geq k$.

11.6 Function overloading

C++ allows you to define multiple functions with the same name, provided the functions have different parameter type lists. This comes in handy when you wish to have similar functionality for data of multiple types. For example, you might want a function which calculates the `gcd` of not just 2 numbers, but several, say 3 as well as 4. Here is how you could define functions for doing both, in addition to the `gcd` function we defined earlier.

```
int gcd(int p, int q, int r){
  return gcd(gcd(p,q),r);
}

int gcd(int p, int q, int r, int s){
  return gcd(gcd(p,q),gcd(r,s));
}
```

The above functions in fact assume that the previous `gcd` function exists. Here is another use. You migth want to have an absolute value functions for `float` data as well as `int`. C++ allows you to give the name `Abs` to both functions.

```
int Abs(int x){
  if (x>0) return x;
  else return -x;
}

float Abs(float x){
  if (x>0) return x;
  else return -x;
}
```

While it is convenient to have the same name in both cases, you may wonder how does the compiler know which function is to be used for your call. The answer is simple: if your call is `abs(y)` where `y` is `int` then the first function is used, if the type is `float` then the second function is used. Likewise, the right `gcd` function will be picked depending upon how many arguments you supplied.

11.7 Function templates

You might look at the two `abs` functions we defined in the preceding section and wonder: sure the functions work on different types, but the bodies are really identical, could we not just give the body once and then have the compiler make copies for the different types? It turns out that this can also be done using the notion of *function templates* as follows.

A function template does not define a single function, but it defines a template, or a scheme, for defining functions. The template has a certain number of variables: if you fix the values of the variables, then you will get a function! Here is an example.

```
template<typename T>
T Abs(T x){
  if (x>0) return x;
  else return -x;
}
```

The name `T` is the template variable. You can put in whatever value you want, and it will define a function. In fact C++ will put in the value as needed! So if you have the following main program

```
int main(){
  int x=3;
  float y=-4.6;

  cout << Abs(x) << endl;
  cout << Abs(y) << endl;
}
```

Then C++ will create two `Abs` functions for you. On seeing the first `cout` statement, C++ will realize that it can create an `Abs` function taking a single `int` argument by setting `T` to `int`. Likewise `T` will be set to `float` and another function will be generated for use in the last statement.

A more interesting example is given in the exercises.

We finally note that the function template must be present in every source file in which a function needs to be created from the template. So a template is best written in a header file. Note also that the template itself cannot be compiled; only the function generated from the template is compiled. So for a template function `f` we will typically have a header file `f.h`, but no file `f.cpp`.

11.8 Exercises

1. Write the function `read_marks_into` and the main program for mark averaging using pointers.

2. A key rule in assignment statements is that the type of the value being assigned must match the type of the variable to which the assignment is made. Consider the following code:

   ```
   int *x,*y, z=3, b[3];

   x = &b;
   y = &x;
   z = y;
   y = *x;
   y = *b;
   ```

   Each of the assignments is incorrect. Can you guess why? If not, write the code in a program, compile it, and the compiler will tell you!

3. Write a function to find roots of a function `f` using Newton's method. It should take as arguments pointers to `f` and also to the derivative of `f`.

4. The k-norm of a vector (x, y, z) is defined as $\sqrt[k]{x^k + y^k + z^k}$. Note that the 2-norm is in fact the Euclidean length. Indeed, the most commonly used norm happens to be the 2 norm. Write a function to calculate the norm such that it can take k as well as the vector components as arguments. You should also allow the call to omit k, in which case the 2 norm should be returned.

5. The function passed to the `bisection` function took a `float` and returned a `float`. However, we might well need to find the root of a function which takes a `double` and returns a `double`. Also, it would be nice if the types of the other arguments were likewise made `double`. Turn `bisection` into a template function so that it works for both `double` and `float` types. You can of course also do this by overloading the name `bisection`.

Chapter 12

Arrays

Real life problems deal with many objects. For example, here are some real life questions that we may be called upon to answer:

- Given the positions, velocities and masses of stars, determine their state 1 million years from today.
- Given the marks obtained by students in a class, print out the marks in decreasing order, i.e. the highest marks first.
- Given the road map of India find the shortest path from Varanasi to Buldhana.

If we want to write programs to solve such problems using what we have learned so far, we would have to separately define variables for each star/student/road. We might want to work with thousands of stars or hundreds of students or roads. Even writing out distinct names for variables to store data for each of these entities will be tiring.

Most programming languages including C++ provide the notion of an *array* so that we can conveniently and tersely deal with large collections of objects.

12.1 Array: Collection of variables

C++ allows us to write statements such as:

```
int abc[1000];
```

This effectively causes 1000 variables to be defined! The first of these is referred to as `abc[0]`, next as `abc[1]`, and so on till `abc[999]`. The collection of these 1000 variables is said to constitute the array named `abc`, and `abc[0]`, `abc[1]`, ..., `abc[999]` are said to constitue the *elements* of the array. Any identifier (Section 3.1.3) can be used to name an to array. What is inside `[ ]` is said to be the *index* of the corresponding element. The term *subscript* is also used instead of index. It is important to note that indices start at 0, and not at 1 as you might be inclined to assume. The largest index is likewise one less than the total number of elements. The total number of elements (1000 in the above example) is referred to as the *length* or the *size* of the array. As we know, an `int` variable needs one word of space, so the statement above reserves 1000 words of space in one stroke.

The space for an array is allocated contiguously, and consecutively by the index, i.e. `abc[1]` is stored in memory following `abc[0]`, `abc[2]` following `abc[1]` and so on.

You may define arrays of other kinds also, e.g.

```
float b[500];  // array of 500 float elements.
```

You can mix up the definitions of ordinary variables and arrays, and also define several arrays in the same statement.

```
double c, x[10], y[20], z;
```

This statement defines variables `c, z` of type `double`, and two arrays `x, y` also of type double and respectively having lengths 10, 20. Note that one variable of type `double` requires 2 words of space, so this statement is reserving 2 words each for `c, z`, and respectively $2 \times 10, 2 \times 20$ words for `x,y`.

You may define arrays in the main program or inside functions as you wish. Note however, that variables defined inside functions are not accessible once the function returns. This applies to arrays defined in functions as well.

12.1.1 Array element operations

Everything that can be done with a variable can be done with elements of an array of the same type.

```
int a[1000];
cin >> a[0];  // reads from keyboard into a[0]

a[7] = 2;     // stores 2 in a[7].

int b = 5*a[7]; // b gets the value 10.

int d = gcd(a[0],a[7]); // gcd is a function as defined earlier.

a[b*2] = 234;   // index: arithmetic expression OK
```

In the first statement after the definition of `a`, we are reading into the zeroth element `a[0]` of `a`, just as we might read into any ordinary variable. But you can set the value of a variable also by assigning to it, as in the statement `a[7]=2;`. The statement following that, `b=5*a[7];` uses the element `a[7]` in an expression, just as you might use an ordinary variable. This is also perfectly fine. Note however, that just like ordinary variables, an element must have a value before it is used in an expression. In other words, it would be improper in the above code to write `int b = 5*a[8];` because `a[8]` has not been assigned a value.

Elements of an array behave like ordinary or *scalar* variables of the same type; so they can be passed to functions just like scalar variables. Hence we can write `gcd(a[0],a[7]);` if we wish, assuming *gcd* is a function taking two *int* arguments.

In the last line in the code the index is not given directly as a number, but instead an expression is provided. This is acceptable. When the code is executed, the value of the expression will be computed and will be used as the index. In the present case, by looking at the preceding code we know that b will have the value 10, and hence a[b*2] is simply a[20]. So 234 will be stored in a[20].

12.1.2 Acceptable range for the index

When using arrays in your programs, it is very important to keep in mind that the array index must always be between 0 (inclusive) and the array size (exclusive). For example, for the array a are defined above, a reference a[1000] would be incorrect, because it is not in the range 0 to 999. Likewise, a reference a[b*2000] would also be incorrect, because it is really the reference a[20000] given that b has value 10 in the code above.

If your program contains such references, C++ does not guarantee how it will behave. It may generate wrong values, fail to terminate, or terminate with an error message. Any one of these outcomes is possible, and C++ does not say which will happen.

Simply put: it is vital that you, the programmer make sure that array indices are in the required range. This is an extremely important requirement.

12.1.3 Initializing arrays

It is possible to combine definition and initialization. Suppose we wish to create a 5 element float array called pqr containing respectively the numbers 15, 30, 12, 40, 17. We could do this as follows.

```
float pqr[5] = {15.0, 30.0, 12.0, 40.0, 17.0};
```

In fact, an alternate form is also allowed and you may write:

```
float pqr[] = {15.0, 30.0, 12.0, 40.0, 17.0};
```

in which the size of the array is not explicitly specified, and it is set by the compiler to the number of values given in the initilizer list. You can of course mix definitions of arrays with or without initialization, and also the definition of variables.

```
int x, squares[5] = {0, 1, 4, 9, 16}, cubes[]={0, 1, 8, 27};
```

This will create a single int variable x, and two initialized arrays, squares of length 5, and cubes of length 4.

Of course, it might be more convenient to initialize arrays separately from their definitions, especially if they are large. So if we wanted a large table of squares, it might be more convenient to write:

```
int squares[100]
for (int i=0; i<100; i++)
   squares[i] = i * i;
```

12.2 Examples

The common use of arrays is to store values of the same type, e.g. velocities of particles, marks obtained by students, lengths of roads, times at which trains leave, and so on. You could also say that an array is perfect to store any sequence $x_1, x_2, \ldots, x_n$. Note the slight peculiarity of C++: it is better to name the sequence starting with 0, i.e. call it $x_0, x_1, \ldots, x_{n-1}$, and then store x_i in ith element of a length n array. As will be discussed in Section 13.1, an array can be used to store text. An array can also be used to store a machine language program: the ith element of the array storing the ith word of the program (Section 2.7). We will see many such uses in the rest of this chapter and the following chapters.

In this section we give some typical examples of programs that use arrays. You will see some standard programming idioms for dealing with arrays.

12.2.1 Notation for subarrays

It will be convenient to have some notation to indicate subarrays of an array. Thus, we will use the notation A[i..j] to mean elements A[k] of the array A where $i \leq k$ and $k \leq j$. Note that if $i > j$, then the subarray is empty.

This notation is only for convenience in discussions, it is not supported by C++ and cannot be used in programs.

12.2.2 A marks display program

Suppose a teacher wants to announce the marks the students in a class have got. One way would be to put up a list on the school notice board. Another possibility is as follows. The teacher loads the marks onto a computer. Then any student that wants to know his marks types his roll number, and the computer displays the marks.[1] Can we write a program to do this?

For simplicity, let us assume that there are 100 students in the class, and their roll numbers are between 1 and 100. Let us also stipulate that the program must print out the marks of each student whose roll number is entered, until the value -1 is supplied as the roll number. At this point, the program must halt.

Clearly we should use an array to store the marks. It is natural to store the marks of the student with roll number 1 in the 0th element of the array, the marks of student with roll number 2 in the first element, and in general, the marks of the student with roll number i in the element at index $i - 1$. So we can define the array as follows.

```
float marks[100]; // marks[i] stores the marks of roll number i+1.
```

You are probably wondering whether we need to change the program if the number of students is different. Hold that thought for a while, we will discuss this issue in Section 12.8.

Next we read the marks into the appropriate array elements.

[1] Many might not like the idea of displaying marks in public. An exercise asks you to add a password so that each student can only see her marks.

```
for(int i=0; i<100; i++){
  cout << "Marks for roll number " << i+1 << ": ";
  cin >> marks[i];
}
```

Remember that when the statement `cin >> marks[i];` is executed, the then current value of `i` is used to decide which element gets the value read. Thus in the first iteration of the loop `i` will have the value 0, and so what is read will be stored in `marks[0]`. In the second iteration `i` will have the value 1 and so the newly read value will be stored in `marks[1]`, and so on. Thus indeed we will have the marks of a student with roll number `i+1` be stored in `marks[i]` as we want.

In the last part of the program, students enter their roll numbers and we are to print out the marks for the entered roll number. Since this is to happen till -1 is given as the roll number, we clearly need a while loop. There are various ways to do this, we choose one with a `break`, similar to Section 6.3

```
while(true){
  cout << "Roll number: ";
  int rollNo;
  cin >> rollNo;

  if(rollNo == -1) break;

  cout << "Marks: " << marks[rollNo-1] << endl;
}
```

Clearly, if you typed 35 in response to the query "Roll number: ", then you would want the marks for roll number 35, and these would be stored in `marks[34]`. But this is exactly the same element as what is printed, `marks[rollNo-1]`.

The program given above will work fine, so long as the roll number given is either -1 or in the range 1 through 100. If a number other than these is given, say 1000, the program will attempt to read `marks[999]`. As we said, this may result in some irrelevant data to be read, or worse, the program may actually halt with an error message. Halting is not acceptable in this situation, because students coming later will then not be able to know their marks. Fortunately we can easily prevent this. If the roll number is not in the given range, then we can say so and not print any marks. So the code should really be as follows.

```
while(true){
  cout << "Roll number: ";
  int rollNo;
  cin >> rollNo;

  if(rollNo == -1) break;

  if(rollNo < 1 || rollNo > 100)
    cout << "Invalid roll number." << endl;
  else
```

```
    cout << "Marks: " << marks[rollNo-1] << endl;
  }
```

12.2.3 Who got the highest?

Having read in the marks as above, suppose we wish to print out the roll numbers of the student(s) who got the highest marks, instead of answering student marks queries.

What we want can be done in 2 steps. In the first step we determine the maximum marks obtained. In the second, we print out the roll numbers of all who got the maximum marks.

In Section 3.4.1 we have already discussed how to find the maximum of the numbers read from the keyboard. Now instead of getting the marks from the keyboard, we are required to read them from the array. Basically, instead of reading from the keyboard, the first element will be obtained from `marks[0]`, and subsequent elements by looking at `marks[i]`, where `i` has to go from 1 to 100. The code for this is as follows.

```
float maxSoFar = marks[0];
for(int i=1; i<100; i++){   // i starts at 1 because we already took marks[0]
  if(maxSoFar < marks[i])
    maxSoFar = marks[i];
}
```

The next step is to print the roll numbers of those students who got marks equal to `maxSoFar`. This is easily done, we examine each `marks[i]`, for all `i` as `i` goes from 0 to 99, and whenever we find `marks[i]` equalling `maxSoFar`, we print out `i+1`, because we stored the marks of roll number `i+1` at index `i`.

```
for(int i=0; i<100; i++)
   if(marks[i] == maxSoFar)
     cout << ‘‘Roll number ‘‘ << i << ‘‘ got maximum marks.’’ << endl;
```

12.2.4 General roll numbers

In the code above, we exploited the fact that the roll numbers are consecutive. In general this may not happen. Often, the roll number assigned to each student may encode different kinds of information, e.g. first two digits are year of joining, another digit indicates the department to which the student belongs, and so on. Sometimes the roll number may also contain letters, though for simplicity we will ignore this possibility.

We consider the marks display problem in this new setting. We will use an additional array `rollno` in which to store the roll number, in addition to the array `marks` used above. The teacher first types in 100 pairs of number, each pair consisting of a roll number and the marks obtained by the student having that roll number. Our program must read in the roll number and marks and store them in the arrays `rollno` and `marks`. In the second phase, when a student types in a roll number, we must first look for it in the array `rollno`. If it is found, then we print the corresponding marks.

```
  int rollno[100];
  double marks[100];
```

```
for(int i=0; i<100; i++) cin << rollno[i] << marks[i];

while(true){
  int r; cin >> r;  // roll number whose marks are requested
  if(r == -1) break;
  for(int i=0; i<100; i++)
     if(rollno[i] == r) cout << marks[i] << endl;
}
```

This idea, scanning an array from the beginning to the end in order to determine if a certain element is stored in the array, is sometimes called *linear search*.

The code above is unsatisfactory in two ways. First, if the given value `r` is not present in the array, it would be polite to print a message to that effect. Second, once we find `r` at some index, there is no need to scan the remaining elements. Both these goals can be acheived by replacing the `for` loop above with the following.

```
int i;
for(i = 0; i<100; i++){
   if(rollno[i] == r){ cout << marks[i] << endl; break;}
}
if(i >= 100) cout << "Invalid roll number.\n";
```

Note first that we break out of the loop upon finding a match. Thus, if a match is found the variable `i` (which has now been defined outside the loop) will have a value less than 100. The check at the end succeeds only if all 100 iterations were executed without finding a match, i.e. if the roll number `r` is invalid.

12.2.5 Histogram

Our next example is trickier, and it illustrates an important powerful aspect of arrays.

Again, we have as input the marks of students in a class. Assume for simplicity that the marks are in the range 0 through 99. We are required to report how many students got marks between 0 and 9, how many between 10 and 19, how many between 20 and 29 and so on. As you might know, what we are asked to report is often called a *histogram* in Statistics[2].

We are required to report 10 numbers, the count of the students receiving marks between 0 and 9, between 10 and 19 and so on till the count of the numbers receiving marks between 90 and 99. So it could seem natural to use an array of 10 elements. The 0th element of the array should count for the range 0-9, the first element for the range 10-19 and so on. So in general we could say ith element of the array should correspond to the range `i*10` to `(i+1)*10-1` (both inclusive). So we could call it `count` and define it as:

[2] In general a histogram is a count of number of observations (marks, in our case) falling in various ranges of values (in our case the intervals 0-9, 10-19 and so on). The counts are often depicted as a bar chart, in which the height of the bars is proportional to the count and width to the range.

```
int count[10]; // count[i] will store the number of marks in the range
               // i*10 through (i+1)*10 -1.
```

Clearly, we should set the counts to 0 at the beginning, and change them as we read in the marks.

```
for(int i=0; i<10; i++)
  count[i]=0;
```

When we read the next mark, how do we decide which count to increment? It is natural to write something like the following.

```
  for(int i=0; i< 100; i++){
    float marks;
    cin >> marks;
    if(marks <= 9) count[0]++;
    else if(marks <= 19) count[1]++;
    else if(marks <= 29) count[2]++;
    else if(marks <= 39) count[3]++;
    else if(marks <= 49) count[4]++;
    else if(marks <= 59) count[5]++;
    else if(marks <= 69) count[6]++;
    else if(marks <= 79) count[7]++;
    else if(marks <= 89) count[8]++;
    else if(marks <= 99) count[9]++;
    else cout << "Marks are out of range." << endl;
  }
```

This works, but there is a better way! Suppose we read a mark m, which count should we increase? For this we simply need to know the tens place digit of m. As you might observe, this is simply $\lfloor m/10 \rfloor$, i.e. the integer part of $m/10$. But we can get the integer part by storing into an integer variable! Which is what the following code does.

```
  for(int i=0; i< 100; i++){
    float marks;
    cin >> marks;
    int index = marks/10;
    if(index >= 0 && index <= 9) count[index]++;
    else cout << "Marks are out of range." << endl;
  }
```

Note that this works only because all the ranges are of the same size. But this is very often the case when computing histograms.

12.2.6 A taxi dispatch program

Suppose you are the Mumbai dispatcher for the Mumbai-Pune taxi service. Your job is as follows. Drivers of taxis that are willing to take passengers to Pune report to you and give you their driver ID number and wait. Passengers who want taxis also report to you. When a passenger reports, you check if there are any waiting taxis. If there are, you assign the taxi of the driver that reported to you the earliest. Clearly, once a taxi has been given to a passenger, you need not keep the corresponding ID number on your list. If no taxis are available, you let the passenger know. You are not expected to keep track of waiting passengers, though an exercise asks you to do precisely this. You may assume that at any given point there will not be more than 100 taxis waiting for passengers. You are to write a program which will help you dispatch taxis as required.

Clearly, we will need to store the IDs of the waiting drivers. It seems natural to use an array, say `driverID`, to store these. Assume for simplicity that the IDs are integers with 9 or fewer digits, i.e. that they will fit in `int`. The size of the array can be a number larger than the number of drivers we expect will be waiting with us at any time. Most of the time there will be fewer drivers waiting with us than the size of the array, so we presumably need a variable `nWaiting` which will denote the number of waiting drivers.

We also need to somehow record the order in which the drivers arrived, because we want to assign the next customer to the driver who has registered with us the earliest. A natural way to do this is to store the earliest waiting driver at index 0, the next earliest at index 1, and so on. The ID of the driver that arrived last would be at index `nWaiting - 1`. If a new driver arrives, we can store his ID at the index `nWaiting`, and increment `nWaiting`. If a customer arrives, we can assign the driver at `driverID[0]`. However, once we assign the driver, we must shift up all the other entries in the array, since we have decided that the waiting drivers must be stored starting at index 0. This scheme will work, but we will see an improvement later.

We must also decide how the program interacts with the rest of the world. Let us suppose that the dispatcher will type ’d’ when a driver arrives, and then the driverID. Likewise when a customer arrives, the dispatcher will type ’c’, and expect the program to print the ID of the assigned driver. Finally, to terminate the program, perhaps we can have the dispatcher type ’x’ (commonly used as abbreviation of eXit).

It is often quite easy to write a program once we decide (a) what our main variables will be and how we plan to use them, and (b) how the user interacts with the program. We merely stick to the rules we have made! If the rules are comprehensive enough, the program almost *writes itself!*

Here is the program.

```
const int n = 100; // estimate of max waiting drivers.
int driverID[n], nWaiting = 0;

while(true){  /* Invariants: nWaiting denotes the number of waiting drivers.
                 0 <= nWaiting <= n.  IDs of waiting drivers are in driverID,
                 from driverID[0] to driverID[nWaiting - 1] */
  char command; cin >> command;
  if(command == ’d’){                               // driver arrives
    if(nWaiting >= n) cout << "Queue full.\n";
    else{
      cin >> driverID[nWaiting];
      nWaiting++;
    }
  }
  else if(command == 'c'){                      // customer arrives
    if(nWaiting == 0) cout << "Nothing available.  Try later.\n";
    else{
      cout << "Assigning " << driverID[0] << endl;
      for(int i=1; i < nWaiting; i++) // shift up waiting drivers
        driverID[i-1] = driverID[i];
      nWaiting--;
    }
  }
  else if(command == 'x') break;
  else cout << "Illegal command.\n";
}
```

Note that we have added checks to see the array `driverID` is already full when a driver is to be entered, and to see if there is at least one element in it when a customer arrives.

You might think that perhaps there should be a way to write the program without having to shift up the entries in `driverID` when we assign the driver at index 0. Instead of moving up the drivers in the array, could we not adjust our notion about where the front of the queue is? Indeed this will work. But it will need a bit more care. To do this right, perhaps it is worth considering how we might have dispatched taxis without computers.

Dispatching without computers

It is always worth thinking about how any problem, including taxi dispatching, might be solved without computers. Say the dispatcher writes the driverID numbers on a blackboard, top to bottom, as the drivers report. When a driver arrives, we put down the number at the bottom of the list. When a passenger comes in, the number at the top of the list is given to the passenger, and that number is erased.

For simplicity, let us assume that our blackboard can only hold 100 phone numbers. Managing this space on the blackboard turns out to be slightly tricky. Suppose 60 drivers report, and you write down their numbers, starting at the top. Suppose you next have 50 passengers, so you match them to the top 50 numbers, which you erase. At this point you have only 10 numbers on the board, however, they are not at the top of the board, but they start halfway down the board. Suppose now 60 more drivers report, . You would place 40 of these numbers below the 10 you have on the board, and that would take you to the bottom of the board. Where should you place the remaining 20? It is natural to start writing numbers from the top again, as if the bottom of the board were joined to the top. Think of the blackboard as forming the curved surface of a cylinder! Thus at this point, you have 70 numbers on the board. They begin at position 50 (the topmost position being 0), go to the last position, 99. Then they "wrap around" so that the last 20 numbers occupy positions 0 through 19 on the board. Positions 20-49 are then unused. This should not confuse us; say we make a mark next to the first waiting driver. When we assign a driver, we erase the number from the board, and also shift the mark down one. This works fine so long as the number of taxi drivers waiting at any time does not exceed 100.

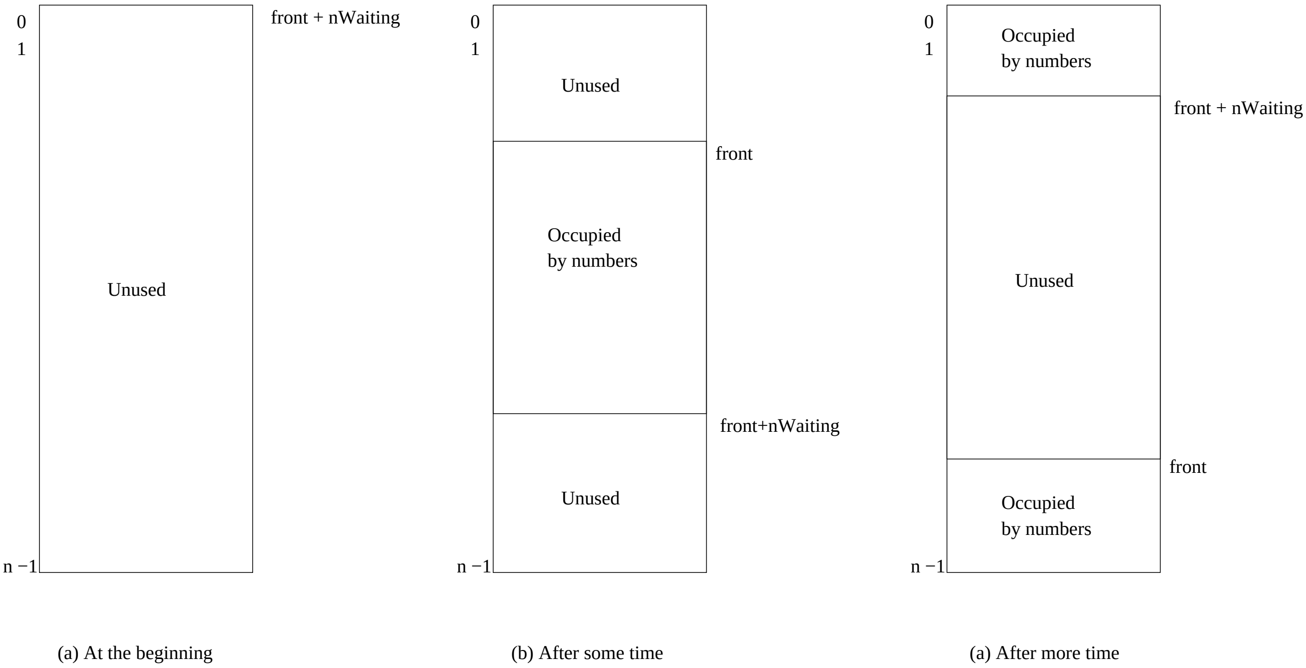


Figure 12.1: Snapshots of the board

Emulating a blackboard on a computer

Our program will mirror the actions given above. We do not shift drivers up when a driver is assigned, instead we have a variable `front`, which will always contain the index of the element of `driverID` containing the earliest waiting unassigned driver. This performs the function of the mark that the dispatcher places on the board.

If there are `nWaiting` drivers waiting for customers, their IDs will appear at positions starting at `front`. We need to be slightly careful as we say this: how do we describe what happens when the board fills to the bottom and the dispatcher is forced to write the numbers starting at the top again? We would like to somehow say that index that comes "next" after the last index `n-1` (i.e. the "bottom " of the board) is is index 0 (i.e. the "top" of the board). Turns out this is easy to say using modulo arithmetic! Thus we will require that if there are `nWaiting` drivers that are waiting, their IDs will be at indices `front`, `(front + 1) % n`, . . ., `(front + nWaiting - 1) % n`.

Next we consider what actions to execute when a driver arrives. As before, we accept the ID only if `driverID` is not full. The ID of the arriving driver must be added at the so as to not violate the property mentioned above, i.e. at position `(front + Waiting) % n`. After that we must increment `nWaiting`.

When a customer arrives, as before we first check if there are any waiting drivers. If there are, we assign the driver at the front of the queue, i.e. `driverID[front]`. We add one to `front` to get to the next element of `driverID`. however, since we want to consider the queue to start again from the top, the addition is done modulo `n`. Finally, we must decrement `nWaiting`.

```
const int n = 100; // estimate of max waiting drivers.
```

```
int driverID[n], nWaiting = 0, front = 0;

while(true){ /* Invariants: nWaiting denotes the number of waiting drivers.
                0 <= nWaiting <= n.  IDs of waiting drivers are in driverID,
                from driverID[front] to driverID[(front + nWaiting - 1) %n ].
                0 <= front < n
             */
  char command; cin >> command;
  if(command == 'd'){                            // driver arrives
    if(nWaiting >= n) cout << "Queue full.\n";
    else{
      cin >> driverID[(front + nWaiting) % n];
      nWaiting++;
    }
  }
  else if(command == 'c'){                       // customer arrives
    if(nWaiting == 0) cout << "Nothing available.  Try later.\n";
    else{
      cout << "Assigning " << driverID[front] << endl;
      front = (front + 1) % n;
      nWaiting--;
    }
  }
  else if(command == 'x') break;
  else cout << "Illegal command.\n";
}
```

12.2.7 A geometric problem

Suppose we are given the positions of the centers of several circles in the plane as well as their radii. Our goal is to determine whether any of the circles intersect. Let us say that the ith circle has center (x_i, y_i) and radius r_i, for $i = 0, \ldots, n-1$.

Whether a pair of circles intersect is easy to check: the circles intersect if and only if the distance between their centers is smaller than or equal to the sum of their radii. In other words, circle i and circle j intersect if and only if:

$$\sqrt{(x_i - x_j)^2 + (y_i - y_j)^2} \leq r_i + r_j$$

Or equivalently $(x_i - x_j)^2 + (y_i - y_j)^2 \leq (r_i + r_j)^2$. Thus, in our program we must effectively check whether this condition holds for any possible i, j, where of course $i \neq j$.

Here is how we can do this. We will use arrays `x,y,r` in which we will store the x,y coordinates of the center and the radius of the circles. Specifically, the x coordinate of the center of the ith circle will be stored in `x[i]`, the y coordinate in `y[i]`, and the radius in `r[i]`. We will then check whether each circle `i` intersects with a circle `j` where `j>i`.

```
  int n=5;                        // number of circles.  5 chosen arbitrarily.
  float x[n], y[n];             // coordinates of center of each circle.
  float r[n];                   // radius of each circle.

  for(int i=0;i<n;i++)          // read in all data.
    cin >> x[i] >> y[i] >> r[i];
                                // Find intersections if any.
  for(int i=0; i<n; i++){
    for(int j=i+1; j<n; j++){
      if(pow(x[i]-x[j],2)+pow(y[i]-y[j],2) <= pow(r[i]+r[j],2))
        // built in function pow(x,y) = x raised to y.
        cout << "Circles " << i << " and " << j << " intersect." <<endl;
    }
  }
```

Thus in the first iteration of the outer for loop, we check for intersections between circle 0 and $1, 2, 3, \ldots, n-1$. In the second iteration, we check for intersections between circle 1 and circles $2, 3, \ldots, n-1$, and so on. Is this clear that we check all pairs of circles in this process? Consider the kth circle and the lth circle, $k \neq l$. Can we be sure that the intersection between them is checked? Clearly, if $k < l$, then in the iteration of the outer for loop in which i takes the value k, we will check intersections with circles $k+1, k+2, \ldots, n-1$. This sequence will contain l because $k < l$. Alternatively, suppose $l < k$. Then consider the iteration of the outer for loop in which i= l. In this iteration we will check the intersection of circle l with circles $l+1, \ldots, n-1$. Clearly k will be in this sequence because $l < k$. Thus in either case we will check the intersection between circle k and circle l, for every k, l.

12.3 The inside story

We would like to clarify some details regarding arrays and array accesses. This will specially be useful for understanding how we define functions for operating on arrays. To make the discussion more concrete, suppose we have the following definitions.

```
int p=5, q[5]={11,12,13,14,15}, r=9;
float s[10];
```

Say each variable of type int is most commonly given 4 bytes of memory, and so is a float. Thus we know the above definitions will cause 4 bytes of memory to be reserved for p, $4 \times 5 = 20$ bytes for q, 4 for r, and $4 \times 10 = 40$ bytes for s. We have also said that the memory given for an array is contiguous. Thus, the memory for q will start at a certain address, say Q, and go on to address $Q+19$. The notion of addresses is as per our discussion in Chapter 2 and Section 11.3. Consistent with this description, Figure 12.3 shows how space might have been allocated for these variables.

Next we consider what happens when during execution we encounter a reference to an array element, e.g. q[expression]. How does the computer know where this element is stored? Of course, first the expression must be evaluated. Suppose its value is some v. Then we know that we want the element of q of index v. But because the elements are stored in order, we also know that the element with index v is stored at $Q + 4v$, where Q

Address	Allocation
$Q-5$	...
$Q-4$	`p`
$Q-3$	
$Q-26$	
$Q-1$	
Q	`q[0]`
$Q+1$	
$Q+2$	
$Q+3$	
$Q+4$	`q[1]`
$Q+5$	
$Q+6$	
$Q+7$	
$Q+8$	`q[2]`
$Q+9$	
$Q+10$	
$Q+11$	
$Q+12$	`q[3]`
$Q+13$	
$Q+14$	
$Q+15$	
$Q+16$	`q[4]`
$Q+17$	
$Q+18$	
$Q+19$	
$Q+20$	`r`
$Q+21$	
$Q+22$	
$Q+23$	
$Q+24$	`s[0]`
$Q+25$	
$Q+26$	
$Q+27$	
$Q+28$	...

Figure 12.2: Possible layout

is the starting address for q. Thus if $v = 3$ then we would want q[3], which is stored from $Q + 12$. In general, the vth element of an array which is stored starting at address A would be at $A + kv$, where k is the number of bytes needed to store a single element.

So the important point is, that to get to an array element, the computer must evaluate the index expression, and even after the expression is evaluated it must perform the multiplication and addition to get the address $A + kv$. This is in contrast to how the computer gets the address for an ordinary variable such as p. In this case, the computer already knows where it stored p, and so it can get to it directly. Do note however that the extra work needed to figure out where the element is stored is independent of the length of the array.

12.3.1 Out of range array indices

Suppose now that our program has a statement q[5]=17;. Going as per the definition above, we would try to store 17 in the int beginning at the address $Q + 20$. Notice that this is outside the range of memory allocated for q. In fact, it is quite possible, as shown in our layout of Figure 12.3, that r is given the memory $Q + 20$ through $Q + 23$. Then the statement q[5]=17 might end up changing r! Likewise it is conceivable that a statement like q[-1]=30; might end up changing p.

Suppose on the other hand, we wrote q[10000]=18;. This would require us to access address $Q + 40000$. It is conceivable that there isnt any memory at this address. Many computers have some circuits to sense if an access is made to a non-existent address or even some forbidden addresses. The details of this are outside the scope of this book, but if this happens, then the program might halt with an error message. In any case, it is most important to ensure that array indices are within the required range.

12.3.2 The array name by itself

So far we have not said whether the name of an array can be used in the program by itself, i.e. without specifying the index. It turns out that C++ allows this.

In C++, the name of an array by itself is defined to have the value equal to the starting address from where the array is stored. Thus the value of q would be Q, and that of s, $Q + 24$, assuming the layout is as per Figure 12.3. Since the variable at address Q is q[0], of type int, it is natural to define the type of q to be pointer to int, or address of int, or int*.

Analogously s would then be of type address of float or pointer to float or float*, and would have the value $Q + 24$.

It seems strange that the name of an array is only associated with the starting address, and that the length of the array is not associated with the name. This is merely a matter of convenience, and its utility will become clear in Section 12.4.

12.3.3 [] as an operator

A further tricky point is that C++ considers a reference to an array element, such as X[Y] to be an expression, with X,Y the operands, and [] the operator![3] The operation is defined only if X has the type "address of some type T", and Y is an expression that evaluates to a value of type int. Suppose that X is of type address of type T, and Y does evaluate to int. Then the expression X[Y] denotes the variable of type T stored at the address $A+kv$, where A is the value of X, v the value of Y, and k is the number of bytes needed to store a single element of type T.

You will realize that we are merely restating how we find the element given the name of the array and the index. But the restatement is more general: X does not need to be the name of an array, it could be any name whose type is "address of some type T". This generalization will come in useful in Section 12.4.

Just to drive home the point, consider the following code.

```
double speed[]={1.25, 3.75, 4.3, 9.2};
double *s;
s = speed;
cout << s[0] << endl;
s[1] = 3.9;
cout << speed[1] << endl;
```

The first point to note is that the assignment s = speed is very much legal, since speed is of type double*, just like s. Thus after the assignment, s will have the same value as speed. But then, the expressions s[j] will mean the same as speed[j].

Put differently, the expression s[0] denotes the double value stored at the address s + k*i, where k is the size of double, and i the value of index, which are respectively 8 and 0. In other words, s[0] means the double stored at address s which is the same as speed, and hence is the same as the value stored at address speed, and so is speed[0]. Thus the first print statement will print 1.25. The statement s[1] is likewise equivalent to speed[1], i.e. to 3.9, which is what the second print statement will print.

12.4 Function Calls involving arrays

Functions are convenient with ordinary, or *scalar* variables, and indeed we can imagine that they will be convenient with arrays as well. Suppose we have an array of floats defined as float a[5]; and somewhere in the program we need to calculate the sum of its elements. Of course we can write code to compute the sum, however, if the sum is needed for several such arrays in our code, then will have to replicate the code that many times. So it would be very convenient to write a function which takes the array as the argument and returns the sum. Here is how the function could be written:

```
float sum(float* v, int n){
```

[3] Yes, this is an unusual way of writing a binary expression. But do note that there are other operations which are not written in the order operand1 operator operand2. For example, we often write $\frac{a}{b}$ rather than $a \div b$.

```
  float s = 0;
  for(int i=0; i<n; i++)
    s += v[i];

  return s;
}
```

This function could be called using a main program as follows.

```
int main(){
  float a[10] = {0.0, 1.0, 2.0, 3.0, 4.0, 5.0},  asum;
  asum = sum(a, 5);  // second argument should be array length
}
```

Let us first check whether the function call is legal, i.e. whether the types of the arguments match those of the parameters in the definition in the function `sum`. The first argument to the function call is the array name `a`. We said that the type associated with the array name is "address of a variable of the type in the definition of the name", in other words the type of `a` is address of a `float`. This indeed matches the type of the first parameter in the function definition. The second argument, 5, clearly has type `int` which matches the type of the second parameter `n`. Thus the call is legal and we now think about how it executes.

When the call `length(a, 5)` is encountered, as usual an area is created for execution of the function `sum`. The values of the non-reference arguments are copied. In the present case, none of the parameters are reference parameters, and so the values of both arguments are copied. The value of the first argument, `a` is the starting address, say A. The value of the second argument is 5. Thus `v` gets the value A and `n` the value 5. It is very important to note here that the content of all the locations in which the array is stored are not copied, but only the starting address is copied.

The code of the function is then executed. The only new part is the expression `v[i]`. This is processed essentially according to the rule given earlier. We know that `v` has type address of float, and its value is A. So now the expression `v[i]` is evaluated as discussed in the previous section, by considering `[]` to be an operator and so on. Instead of doing the precise calculation again, we merely note that the value of `v[i]` evaluated in `sum` must be the same as the value of `a[i]` evaluated in the main program, because `v` has the same value and type as `a`. Hence, `v[i]` will in fact denote the `i`th element of `a`. Because `n` has value 5, in the loop `i` will take values from 0 to 4. Thus `a[0]` through `a[4]` will be added as we desired.

Some remarks are in order.

1. Another syntax can also be used to declare parameters that are arrays, in this case arrays of `float` variables: `float v[]` – this directly suggests that `v` is like an array except that we do not know its length.

2. Function `sum` does not really *know* that it is operating on the entire array. For example the call `sum(a,3)` is also allowed. This would return the sum of the first 3 elements of `a`, since the loop in the function will execute from 0 to 2.

3. The array name is *not* passed as a reference parameter, and so when the call executes, the value of the array name is copied into the corresponding parameter. This has the effect, that using the [] operator the code in the function can refer to the array defined in the main program. We had said earlier that code in the function cannot refer directly to variables defined in the main program. However, because we are passing the address, the code in the function can refer to the array *indirectly*, through the passed address. Modifying the passed array is also possible. If your function had a line at the end such as v[0]=5.0;, that would indeed change a[0]. This is consistent with the mechanism we have discussed for evaluating expressions involving [].

4. If the value of a non-reference parameter to a function is changed in the function, the value of the corresponding argument is not affected, since they are two distinct copies. Thus if you choose to modify v by storing a different address into it (not recommended at all in the present circumstances!), you will not change the value of the corresponding argument a.

12.4.1 Examples

Shown below are two simple examples of functions on arrays. The next sections gives more involved examples.

Our first function merely reads values into a float array.

```
void print(float *a, int n){
  for(int i=0; i< n; i++)
    cout << a[i] << endl;
}
```

This will print out the first n elements of the array. Note that it is the responsibility of the calling program to ensure that the an array is passed, and that the array has length at least as much as the second argument.

Here is a function which returns an index of an element whose value is the maximum of all the elements in the array. Note the careful phrasing of the last sentence: when we say "*an* element", we acknowledge the possibility that there could be many such elements, and we are returning only one of them.

The idea of the function is very similar to what we did for finding the maximum marks from the marks array in Section 12.2.3. We have a variable maxIndex which will return the position of an element with the maximum value. We start by initializing it to 0, which is equivalent to conjecturing that the maximum appears in position 0. Next, we check if the subsequent elements of the array are larger, if we find an element which is larger, then we assign its index to maxIndex.

```
int argmax(float marks[], int length)
// marks = array containing the values
// length = length of marks array.  required > 0.
// returns maxIndex such that marks[maxIndex] is largest in marks[0..length-1]
{
  int maxIndex = 0;
  for(j = 1; j<length; j++)
    if( marks[maxIndex] > marks[j]) // bigger element found?
      maxIndex = j;                 // update maxIndex.
  return maxIndex;
}
```

We have given the name marks so that it is easy for you to see the similarity between this code and the code in Section 12.2.3. But by itself this function does not have anything to do with marks. So if you write it independently some more appropriate name such as datavalues should be used instead of the name marks.

12.4.2 Summary

The most important points to note are as follows.

To pass an array to a function, we must typically pass 2 arguments, the name of the array, and the length of the array. This is to be expected, the name only gives the starting address of the array, it does not say how long the array is. So the array length is needed.

The called function can read or write into the array whose name is sent to it. This is like sending the address of one friend A to another friend B, Surely then B will be able to write to A or visit A just as you can!

Finally, it is worth noting an important point. When we write a function on arrays, it may be convenient to allow it to be called with length specified as 0. What should a function such as sum to when presented with an array of zero length? It would seem natural to return the sum of elements as 0. This is what our sum function does. On the other hand, our argmax function requires that the length be at least 1. Such (pre)conditions on acceptable values of parameters should be clearly stated in the comments.

12.5 Sorting an array

We will consider a problem discussed at the beginning of the chapter: given the list of marks, print them out in the order highest to lowest. In fact we will do this in two ways, which will illustrate some ideas about passing arrays to functions.

We could ask that along with the marks, we also print out the roll numbers, however, this is left for the exercises. In this section, we will ignore the fact that the marks stored at index i are the marks obtained by the student with roll number i. This allows us to use the following two phase strategy.

In the first phase, we rearrange the element values so as to ensure that the values appearing at lower indices are no smaller than those appearing at larger indices. This operation is often called sorting. This is one of the most important operations associated with an array. We will present a simple algorithm for this. Better algorithms will be given in later chapters. Once the elements are arranged so that the larger ones appear before the smaller ones, we can simply print out the array elements by index, i.e. element 0, then element 1 and so on. This will ensure that the marks are printed in non increasing order. For this, we can simply use the function print defined earlier.

We use a fairly natural idea for sorting. We begin by looking for the largest value in the array, and we move it to the position (index) 0 in the array. Of course, position 0 itself contains a value, and we cannot destroy that. So we instead exchange the two: the maximum value moves to the 0th position and the value in the 0th position moves to wherever the maximum was present earlier. Next, we find the maximum value amongst elements in position 1 through $n-1$, where n is the array length. This maximum is exchanged with the element in position 1. Thus we have the maximum and second maximum in positions 0 and 1. We go on in this manner.

The basic operation in our algorithm is then the following. We need to find the largest among the elements in positions i through $n-1$, and exchange that with the element at position i. We will write a function for doing this. The function will have to take as arguments the name of the array, say m, the length n, and the starting index i.

As you can see, this is simply a minor extension of the function argmax that we wrote earlier. There are two differences, we only need to consider the array starting at a given index i, and at the end we must perform the exchange.

```
void largest_of_i_to_last_moves_to_i(float *m, int n, int i)
// This will find a maximum value in the region m[i..n-1] and move
// it to m[i].  The value at m[i] will move to where the maximum was.
{
  int candidate = i; // index of candidate
  int maxSoFar = m[candidate];

  for(int i=1; i < n; i++){
    if(m[i] > maxSoFar){
      maxSoFar = m[i];
      candidate= i;
    }
  }

  marks[candidate] = marks[i];   // exchange the values.
  marks[i] = maxSoFar;
}
```

All that remains now is to use this function. We must call it with different values of i. We can then write the function to sort an array as follows.

```
void sort(float data[], int n)
// will sort the array data of length n in non-increasing order.
{
  for(int i=0; i<n; i++)
     largest_of_i_to_last_moves_to_i(data, n, i);
}
```

We have to be careful in calling the function move_largest_to_i – is the call in the last iteration correct? In the last iteration, the value of i is n-1, i.e. we are asking that the maximum value in the elements of the array starting at data[n-1] through data[n-1] be placed at `data[n-1]`. Fortunately, our function works fine even if `i` and `n` have the same value. So the code given above is not incorrect. However, note that the last call is not necessary, i.e. the check in the `for` loop might as well have been `i<n-1` rather than `i<n`.

Our second method is more subtle. We first present the code, which uses the `argmax` function defined in Section 12.4.1.

```
void sort2(float data[], int n)
// will sort in NON-DECREASING order.  different from above.
{
  for(int i=n; i>1; i--){
    int maxIndex = argmax(data,i);
    float maxVal = data[maxIndex];
    data[maxIndex] = data[i-1];
    data[i-1] = maxVal;
  }
}
```

The most noteworthy point of this code is the call to `argmax`. Even though the length of the array `data` is `n`, we are calling it with successively smaller values. As we discussed earlier, this is acceptable, the function `argmax` will only consider the first `i` elements of the array, in each invocation.

So in the first iteration, we find the index of a largest element. The 3 lines after the call to `argmax` merely exchange the values of the `maxindex`th element (as returned by argmax) and the `i-1`th element, which in the first iteration is simply the last element of the array. Thus at the end of the first iteration, a largest element has moved to the end of the array. In the next iteration, we repeat the process, but with a smaller value of `i`. Thus in the second iteration, we will have moved the second largest element to the position `i-1`, which in the second iteration has value `n-2`. This we need to do until `i` becomes 2, because when `i=1`, `argmax` will be asked to find the maximum of (what it thinks is) an array of length 1, and this is unnecessary.

12.6 Binary search

We often sort data because it looks nice to print it that way. However, sorting helps in performing certain kinds of operations very fast.

Suppose we have an array in which we have stored numbers. Suppose now that we want to determine if a given number x is present in the array. Obviously, we will need to go over each element in the array and check if it equals x. If x is not present, we will know that only after looking at every element. If x is present, it could be at any index in the array. In the worst case we might still have to examine every array element; on the average we could say that we would examine about half the elements.

The situation is dramatically different if the array is sorted. Instead of examining elements from the beginning of the array, in the first step we examine the element that is roughly in the middle of the array. Say our array is `A` and it contains `size` elements. Then in the first step we check if `x < A[size/2]`. Here we mean integer division when we write *size/2*, i.e. the value of `size/2` rounded down. There are 2 cases to consider.

The check succeeds i.e. `x` is smaller than `A[size/2]`. Now because the array is sorted, we know that all elements in the subarray `A[size/2+1..size-1]` will also be larger than `x`. Hence `x`, if present in the array, will be in the portion `A[0..size/2-1]`. Thus using just 1 comparison, we have narrowed our search to the first half of the array.

The check fails i.e. `x` is greater of equal to `A[size/2]`. In this case, we can narrow our search to the second half, i.e. `A[size/2..size-1]`. This can be proved as follows. There are two cases to consider: (i) `A[size/2]` is strictly smaller than `x`. In this case, we know that the values in `A[0..size/2-1]` will be strictly smaller. Thus the subsequent search needs to be made only in `A[size/2..size-1]`. (ii) `A[size/2]` equals `x`. In this case also, we can make the subsequent search in `A[size/2..size-1]`, because this region is known to contain `x`.

Thus in both cases, after one comparison, we have ensured that subsequently we only need to search in one of the halves of the array. But we can recurse on the halves!

The key question is: when does the recursion end. Clearly, if our array has only one element, then we should not try to halve it! In this case we merely check if the element equals `x` and return the result of the comparison.

This gives us the following recursive function.

```
bool Bsearch(int x, int A[], int start, int size){
// x : target value to search
// range to search: A[start..start+size-1]
// precondition: size > 0;
//
  if(size == 1) return (A[start] == x);
  int half = size/2;                 // 0 < half < size, because size>1.
  if(x < A[start+half])
    return Bsearch(x, A, start, half);           // recurse on first half
  else
    return Bsearch(x, A, start+half, size-half); // recurse on second half.
}
```

There is an extra parameter, `start` which says where the subarray starts. So we are searching in the region `A[start...start+size-1]`. The "middle" element now is `A[start+size/2]` which is the same as `A[start+half]` in the code. The "first half" starts at `A[start]` and has size equal to `half`. The "second half" starts at `A[start+half]` and has size `half`. Thus we have the recursive calls in the function.

Here is a main program which tests the function.

```
main_program{
  const int size=10;
  int A[size]={1, 2, 2, 3, 10, 15, 15, 25, 28, 30};
  for(int i=0; i<size; i++) cout << A[i] << " ";
  cout << endl;

  for(int x=0; x<=40; x++)
```

```
    cout << x << ": " << Bsearch(x, A, 0, size) <<endl;
}
```

We search for presence of all integers between 0 and 40. You will see that 1 is returned only for those integers that are actually present in the array.

Notice that the array is sorted, but contains repeated values.

12.6.1 Time required

Let us analyze a bigger example. Suppose we are checking for the presence of a number in an array of size 1024. How many array elements do we compare in the process?

The function `binsearch` will first be called with the `size` parameter equal to 1024. When we recurse, no matter how the comparison comes out, we will next call `binsearch` with `size` 512. Subsequently we call `binsearch` with `size` 256 and so on. Thus a total of 10 calls will be made: in the last call `size` will become 1 and we will return the answer. In each call we make only one comparison `x < A[start+half]`, and hence only 10 comparisons will be made!

Compare this with the case in which the array is not sorted: then we might have to make as many as 1024 comparisons! Even if we agree that it takes a bit longer to call a function, calling `binsearch` 10 times (including the recursion) will be much faster than executing a loop to do 1024 comparisons. Actually, our binary search can be written out as a loop, without recursion, the exercises ask you to do this.

In general you can see that the number of comparisons made is simply the number of times you have to divide the size so as to get the number 1. This number is log(`size`). For the unsorted case we might make as many as `size` comparisons, which is much larger!

Binary search is a simple but important idea. You will see that it will appear in many places, perhaps slightly disguised, as it did in the Bisection algorithm (Section 7.2) for finding roots.

12.7 Representing Polynomials

A program will deal with real life objects such as stars, or roads, or a collection of circles. It might also deal with mathematical objects such as polynomials. How to represent polynomials on a computer and perform operations on them are therefore important questions.

A polynomial $A(x) = \sum_{i=0}^{i=n-1} a_i x^i$ is completely determined if we specify the coefficients $a_0, \ldots, a_{n-1}$. Thus to represent the above polynomial we will need to store these coefficients. This most conveniently done in an array.[4] We use an array `a` of length n and store a_i in `a[i]`.

Next comes the question of how we operate on polynomials. It is natural to ask; suppose we have two arrays representing two polynomials $A(x), B(x)$. Can we construct the rep-

[4] There is a simple rule here – if a collection of objects is described using one subscript, use a one dimensional array, which is what we have studied so far. If a collection of mathematical object is described using two subscripts, say the entries of a matrix, then we will need two dimensional arrays, which we will see later.

resentation of the polynomials $C(x), D(x)$ obtained by adding and multiplying $A(x), B(x)$ respectively?

We know that $c_i = a_i + b_i$. Thus the array c that we can use to represent the polynomial $C(x)$ must be defined as float c[n]. Further, its value can be assigned using the following code:

```
for(int i=0; i<n; i++)
   c[i] = a[i] + b[i];
```

Can we write a function addp which adds two polynomials? The polynomials to be added will be passed as arguments. What about the result polynomial? We could allocate a new array inside the function addp, but this array cannot be returned back – it gets destroyed as soon as addp finishes execution. The correct way to write this procedure is to pass the result array as well. Here is a program which includes the function addp.

```
void addp(float a1[], float a2[], float r[], int n){
// addends, result, length of the arrays.
  for(int i=0; i<n; i++) r[i] = a1[i]+a2[i];
}

int main(){
  float a[5], b[5],c[5];
  for(int i=0; i<5; i++) cin >> a[i] >> b[i];

  addp(a,b,c,5);

  for(int i=0; i<5; i++) cout << c[i] << endl;
}
```

We will likewise use an array d to represent the product $D(x)$. The product can have $2n-1$ coefficients. Thus it will have to be defined as float d[2*n-1]. To assign values to its elements, we need to first consider how the coefficients of D relate to those of A, B. When $A(x)$ and $B(x)$ are multiplied, each term a_jx^j in the former will be multiplied with b_kx^k in the latter, producing terms $a_jb_kx^{j+k}$. Thus, this will contribute a_jb_k to d_{j+k}. This gives us the program.

```
void prodp(float a[], float b[], float d[], int n){
// a,b must have n elements, product d must have 2n-1.
  for(int i=0; i<2*n-1; i++) d[i] = 0;
  for(int j=0; j<n; j++)
    for(int k=0; k<n; k++) d[j+k] += a[j]*b[k];
}
```

We could write this differently by asking: which products contribute to the term d_ix^i? Clearly, the products of terms a_jx^j and $b_{i-j}x^{i-j}$ will produce $a_jb_{i-j}x^i$ and will thus contribute. Thus we will have:

$$d_i = \sum_j a_j b_{i-j}$$

The question is what should the limits on j be. Clearly, we require $0 \leq j \leq n-1$ so that a_j is well defined and $0 \leq i-j \leq n-1$, so that b_{i-j} is also well defined. From the second we get $i-n+1 \leq j \leq i$. Thus we can conclude that the limits are:

$$d_i = \sum_{j,k,j+k=i} a_j b_k = \sum_{\max(0,i-n+1)}^{\min(i,n-1)} a_j b_{i-j}$$

This can be easily coded as:

```
void prodp(float a[], float b[], float d[], int n){
// a,b must have n elements, product d must have 2n-1.
   for(int i=0; i<2*n-1; i++){
     d[i] = 0;
     for(int j=max(0,i-n+1); j<= min(i,n-1); j++)
       d[i] += a[j]*b[i-j];
   }
}
```

12.8 Array Length and `const` values

In the examples given above, we have explicitly written out numbers such as 500,1000 to specify the array length. Arrays will often be used in programs for storing a collection of values, and the total number of values in the collection will not be known to the programmer. So you might consider it more convenient if we are allowed to write:

```
int n;
cin >> n;
int a[n];  // Not allowed by the standard!
```

This code is not allowed by the C++ standard. The C++ standard requires that the length be specified by an expression whose value is a *compile time constant*. A compile time constant is either an explicitly stated number; or it is an expression only involving variables which are defined to be `const`, e.g.

```
const int n = 1000;
```

The prefix `const` is used to say that `n` looks like a variable, and it can be used in all places that a variable can be used, but really its value cannot be changed. So using a `const` name, arrays might be defined as follows.

```
const int NMAX = 1000; // convention to capitalize constant names.
int a[NMAX], b[NMAX];
```

So how do we use this in practice? Suppose we want to define an array which will store the marks of students. In this case, the C++ standard will require us to guess the maximum number of students we are likely to ever have, define an array of that size, and only use a part of it. So we might write:

```
const int NMAX = 1000;
int a[NMAX], b[NMAX], nactual;
cin >> nactual;
assert(NMAX <= nactual);
```

In the rest of the code, we remember that only the first nactual locations of a and b are used, and so write loops keeping this in mind. Note that it is possible that the user will type in a value for `nactual` that is larger than `NMAX`. In this case we cannot run the program. If this happens, the `assert` statement will cause the program to stop, and you will need to change `NMAX`, recompile and rerun.

12.8.1 Why `const` declarations?

The above code could also directly define `int a[1000],b[1000];` instead of using the `const` definition. However, the code as given is preferable if we ever have to change the required size, say we want arrays of size 2000 rather than 1000. If we had not used `NMAX` we would have to change several occurrences of 1000 to 2000; with the code as given, we just need to change the first line to `const int NMAX = 2000;`

12.8.2 What we use in this book

The GNU C++ compiler that you invoke when you use the command `s++` allows arbitrary expressions to be specified as array lengths in declarations.

As you can see, this makes the code much more compact and easier to understand at a glance. So in the interest of avoiding clutter, in the rest of the book, we will use arbitrary expressions while specifying lengths of arrays. The code we give will work with `s++`. If it does not work for some other compiler, the discussion above tells you how to change it.

12.9 Summary

Arrays provide an easy way to store sets of objects of the same type.

It is worth thinking about how the index of an element gets used. Sometimes the index at which an element is stored has no significance, as in the circle intersection problem. Or sometimes we can make a part of the data be the index, as we did for the roll number in the marks display problem. Similar was the case for the histogram problem. In the taxi dispatch problem, we used the index to implicitly record the arrival order of the taxis.

Suppose we want to look for elements satisfying a certain property. One way to do so is to scan through the array, one element at a time, and check if the element has the required property. We did this in the problem of printing roll numbers of students who had the highest marks. This is a common idiom.

The idea of scanning through the array starting at index 0 and going on to the largest index is also useful when we want to perform the same operation on every element, e.g. print it. We used a somewhat complicated version of this in the circle intersection problem, where we wanted to perform a certain action not for each circle, but for each pair of circles.

Finally, in the taxi dispatch problem we built a so called *queue* so that the elements left the array in the same order that they arrived in. For this we maintained two indices: where the next element will be stored and which element will leave next. This is a very common idiom, and you will see it, for example, in Exercise 14.

12.10 Exercises

1. Suppose the roll numbers in the class do not go from 1 to the maximum number of students, but are essentially arbitrary numbers (because perhaps they identify the year in which the student enters, or the program that the student belongs to, and so on). Write the marks display program for this case. Assume that for each student first the roll number is typed in, and then the roll number. Also assume that at the beginning the number of students is given.

2. Write the program to display who got the maximum marks for the case above, i.e. when the roll numbers are arbitrary integers.

3. Suppose we want to find a histogram for which the width of the intervals for which we want the counts are not uniform. Say each value is a real number between 0 (inclusive) and 1 (exclusive). Between 0 and 0.25, our intervals are of width 0.05, i.e. we want a count of how many values are between 0 and 0.05, then 0.05 and 0.1, and so on. Between 0.25 and 0.75 our intervals are of width 0.025, i.e. we want to know how many values are between 0.25 and 0.275, then 0.275 and 0.3, and so on. Finally, between 0.75 and 1, our intervals are of width 0.05. Write a program that provides the histogram for these ranges.

4. You are to write a program which takes as input a sequence of positive integers. You are not given the length of the sequence before hand, but after all the numbers are given, a -1 is given, so you know the sequence has terminated. You are required to print the 10 largest numbers in the sequence. Hint: use an array of length 10 to keep track of the numbers that are candidates for being the top 10.

5. Suppose in the previous problem you are asked to report which are the 10 highest values in the sequence, and how frequently they appear. Write a program which does this.

6. Suppose we are given the x, y coordinates of n points in the plane. We wish to know if any 3 of them are collinear. Write a program which determines this. Make sure that you consider every possible 3 points to test this, and that you test every triple only once. The coordinates should be represented as `float`s. When you calculate slopes of line segments, because of the floating point format, there will be round-off errors. So instead of asking whether two slopes are equal, using the operator `==`, you should check if they are approximately equal, i.e. whether their absolute difference is small, say 10^{-5}. This is a precaution you need to take when comparing floating point numbers. In fact, you should also ask yourself whether the slope is a good measure to check collinearity, or whether you should instead consider the angle, i.e. the arctangent of the slope.

7. Write a program which takes as input two vectors (as defined in mathematics/physics) – represent them using arrays – and prints their dot product. Make this into a function.

8. Suppose you are given the number n of students in a class, and their marks on two subjects. Your goal is to calculate the correlation. Let x_i, y_i denote the marks in the two subjects. Then the correlation is defined as:

$$\frac{n \sum x_i y_i - \sum x_i \sum y_i}{\sqrt{n \sum x_i^2 - (\sum x_i)^2} \sqrt{n \sum y_i^2 - (\sum y_i)^2}}$$

   Write a program that calculates this. Note that a positive correlation indicates that x increases with y (roughly) – whereas negative correlation indicates that x increases roughly as y decreases. A correlation around 0 will indicate in this case (and often in general) that the two variables are independent. You may use the dot product function you wrote for the previous exercise.

9. Suppose you are given the maximum temperature registered in Mumbai on March 21 of each year for the last 100 years. You would like to know whether Mumbai has been getting warmer over the years, as is generally believed. You would like to know from your data whether this might be a reasonable conclusion. If you merely plot the data, you will see that the temperatures fluctuate apparently erratically from year to year. The weather is expected to behave somewhat randomly; what you want to know is whether there is any upward trend if you can somehow throw out the randomness.

   One way to reduce the randomness is to smooth the data by taking so called *moving averages*. Given a sequence of numbers $x_1, \ldots, x_n$, a $2k+1$-window size moving average is a sequence of numbers $y_{k+1}, \ldots, y_{n-k}$, where y_i is the average of $x_{i-k}, \ldots, x_{i+k}$. Write a program which takes a sequence and the integer k as input, and prints out the $2k+1$ window-size moving average. Also plot the original sequence and the moving average.

10. A sequence $x_0, \ldots, x_n$ (note that the length is $n+1$) is said to be a palindrome if $x_i = x_{n-i}$ for all i. Write a program which takes a sequence whose length is given first and says whether the sequence is a palindrome.

11. The *Eratosthenes' Sieve* for determining whether a number n is prime is as follows. We first write down the numbers $2, \ldots, n$ on paper (or a clay tablet if we can get it!). We then start with the first uncrossed number, and cross out all its proper multiples. Then we look for the next uncrossed number and cross out all its proper multiples and so on. If n is not crossed out in this process, then it must be a prime. Write a program based on this idea. Earlier in the course we had a primality testing algorithm which checked whether some number between 2 and $n-1$ divided n. Is the new method better than the old one?

12. Suppose we are given an array `marks` where `marks[i]` gives the marks of student with roll number `i`. We are required to print out the marks in non-increasing order, along with the roll number of the student who obtained the marks. Modify the sorting algorithm developed in the chapter to do this. Hint: Use an additional array `rollNo`

such that `rollNo[i]` equals `i` initially. As you exchange marks during the course of the selection sort algorithm, move the roll number along with the marks.

13. Suppose you are given a sequence of numbers, preceded by the length of the sequence. You are required to sort them. In this exercise you will do this using the so called *Insertion sort* algorithm. The idea of the algorithm is to read the numbers into an array, but keep the array sorted as you read. In other words, after you read the first i numbers, you must make sure that they appear in the first i elements of the array in sorted (say non-increasing) order. So when you read the $i+1$th number, you must find where it should be inserted. Suppose you discover that it needs to be placed between the numbers that are currently at the jth and $j+1$th position, then you should move the numbers in positions $j+1$ through $i-1$ (note that the indices or positions start at 0) forward in the array by 1 step. Then the newly read number can be placed in the $j+1$th position. Write the program that does this.

14. Suppose you are given two arrays `A,B` of lengths `m,n`. Suppose further that the arrays are sorted in non-decreasing order. You are supposed to fill an array `C` of length `m+n` so that it contains precisely the same numbers which are present in `A,B`, but they must appear in non-decreasing order in `C`. In other words, if `A` contains the sequence 1,2,3,3,5 and `B` contains 2,4,6,7, then `C` should contain 1,2,2,3,3,4,5,6,7. Hint: Clearly, elements must move out of `A` and `B` into `C`. Can you argue that for the purpose of this movement all arrays behave like queues, i.e. elements always move out from the front of `A,B`, and move into the back of `C`?

15. A friend ("the magician") shows you a deck of cards. He picks up the top card, turns it face up, and it is seen to be the ace. He puts the card away. He then takes the next card and puts it at the bottom of the deck without showing it to you. Then he shows you the card now at the top of the deck, which turns out to be the 2. He repeats the process: showing you the top card, keeping it aside, and then moving a card from the top to the bottom without showing it to you. It turns out (magically!) that you see the cards in increasing face value, i.e. the first card to be exposed is the ace, then the 2, then the 3, then the 4, and so on until the King. Of course, the "magic" is all in the order in which the cards were placed in the deck at the beginning. Write a program that explains the magic, i.e. figures out this order? Hint: Reverse the process.

16. Write a function which given polynomials $P(x), Q(x)$ returns their *composition* $R(x) = P(Q(x))$. Say $P(x) = x^2 + 3x + 5$ and $Q(x) = 3x^2 + 5x + 9$. Then $R(x) = (3x^2 + 5x + 9)^2 + 3(3x^2 + 5x + 9) + 5$.

17. Write the binary search code without recursion.

18. Consider a railway track consisting of some n segments. For each ith segment, you are given its length L_i and a maximum speed s_i with which trains can run on it. The maximum speed for different segments can be different, and it depends upon the quality of rails used, whether the track has turns, and other factors. You are also given the data for a certain locomotive: its maximum speed s and the maximum accereration a it is capable of (assume this is independent of the speed, for simplicity), the maximum

deceleration d (again independent of the speed) it is capable of. Suppose the train starts at rest at one end of the track and must come to rest at the other end. How quickly can the train complete this journey? Make sure your code works for all possible values of the parameters.

Chapter 13

More on arrays

We begin by considering the problem of representing textual data. In chapter 3 we discussed the `char` datatype for storing characters. However, we rarely work with single characters. More often, we will need to manipulate full words, or strings/sequences of characters. A character string is customarily represented in C as an array of characters. This is not quite recommended in C++, as we will study later. But it is worth knowing this representation because the C++ recommended representation builds upon this.

Next, we discuss *multidimensional arrays.* An ordinary (one dimensional) array can be thought of as a sequence of values. A two dimensional array can be thought of as a table (rows and columns) of values. It turns out that the standard, built-in way of representing multidimensional arrays in C++ is somewhat inconvenient. However, C++ allows us to build our own, more convenient mechanism. We present one such mechanism which is a part of `simplecpp`. In this chapter we only discuss how to use this mechanism; how it is built is described in Chapter **??**.

We discuss a number of applications of two dimensional arrays, including that of finding shortest paths on a map.

13.1 Character strings

An array of characters can be defined just as you define arrays of `double`s or `int`s.

```
char name[20], residence[50];
```

The above defines two arrays, `name` and `residence` of lengths 20 and 50, ostensibly for storing the name and the residence. Since we will usually not know the exact number of characters in a name or in an address, it is customary to define arrays of what we guess might be the largest possible length. This might seem wasteful, and it is, and we will see better alternatives in later chapters.

So if we want to store a character string “Shivaji” in the array, we will be storing ’S’ in `name[0]`, ’h’ in `name[1]` and so on. The string is 7 characters long, and you would think that we should store this length somewhere. While printing the string for example, we clearly do not want the `name[7]` through `name[19]` printed. The convention used in the C language, and inherited into C++ from there, is that instead of storing the length explicitly, we store a special character at the end of the actual string. The special character used is the one with ASCII value 0, and this can be written as '\0'. Note that '\0' is not printable, and is not expected to be a part of any real text string. So it unambiguously marks the end of the string.

Special constructs are provided for initializing character arrays. So indeed we may write

```
char name[20] = "Shivaji";
char residence[50] = "Main Palace, Raigad";
```

The character string "Shivaji" has 7 characters. So these will be placed in the first 7 elements of `name`. The eighth element, `name[7]` will be set to `'\0'`. Similarly only 20 elements of `residence` will be initialized, including the last `'\0'`. Note by the way that capital and small letters have different codes.

Here is an alternative form.

```
char name[] = "Shivaji";
char residence[] = "Main Palace, Raigad";
```

In this, C++ will calculate the lengths of `name` and `residence`. Following the previous discussion, these will be set to 8 and 20 respectively.

We can manipulate strings stored in `char` arrays by going over the elements in a for loop, for example.

13.1.1 Output

Printing out the contents of a character array is simple. Assuming `name` is a character array as before,

```
cout << name;
```

would cause the contents of `name` from the beginning to the '\0' character to be printed on the screen. It is the responsibility of the programmer to ensure that the array being printed indeed contains a '\0' character.

The general form of the above statement is:

```
cout << charptr;
```

where `charptr` is an expression which evaluates to a pointer to a `char` type. If `name` is a character array, then `name` indeed is of type pointer to `char`. This statement causes characters starting from the address `charptr` to be printed, until a '\0' character is encountered. Thus character arrays passed to functions can be printed in the expected manner.

13.1.2 Input

To read in a string into a `char` array you may use the analogous form:

```
cin >> charptr;
```

Here charptr could be the name of a `char` array, or more generally, an expression of type pointer to `char`. The statement will cause a whitespace delimited string typed by the user to be read into the memory starting at the address denoted by `charptr`. After storing the string the string the '\0' character will be stored. There are a couple of points to be noted, which we explain with an example. Consider the following code.

```
char name[20];
cout << "Please type your name: ";
cin >> name;
```

The second statement asks you to type your name and the third, `cin >> name;` reads in what you type into the array `name`. Two points are worth noting:

1. From what you type, the initial whitespace characters will be ignored. The character string starting with the first non-whitespace character and ending just before the following whitespace character will be taken and placed in `name`. Thus if I type

   ```
       Abhiram Ranade
   ```

   with some leading whitespace, the leading whitespace will be dropped and only `"Abhiram"` would go into `name`. Next, following `"Abhiram"` a null character, i.e. '\0' would be stored. Thus the letters `'A'` through `'m'` would go into `name[0]` through `name[6]`, and `name[7]` would be set to '\0'. Note that all this is very convenient in that the a single statement reads in all the characters, and further the '\0' is also automatically placed after the last character read in.

2. This statement is potentially unsafe. If a user types in more characters than the length of the array, with no whitespace characters in between them, then the characters typed in will be written to the area of memory starting with `name[0]`. In other words, if the user types more characters than the length of `name`, all those will also be stored, possibly damaging the memory following the array `name`. In other words, this is an error of exceeding the size of the array.

The safe alternative to this is to use the following command.

```
cin.getline(x,n);
```

where `x` must be a name of a `char` array (or more generally a pointer to `char`), and `n` an integer. This will cause whatever the user types, including whitespace characters, to be placed into the array `x`, until one of the following occurs

- A newline character is typed by the user. In this case all characters upto the newline are copied into *x*. The newline character is not copied. It is discarded.

- `n-1` characters are typed without a newline. In this case all the characters are placed into `x`, followed by a '\0' character.

As you may guess, it is customary to use the length of `x` as the argument `n`. So for example we can write:

```
char name[20];
cin.getline(name,20);
```

In this case at most 19 characters that the user types will be copied, and we will have no danger of overflowing past the array limit.

13.1.3 Character string constant

Quoted text, such as `"Please type your name:"` constitutes a *string constant* in C++. The compiler stores the string somewhere in memory (followed by '\0'), and you may refer to it. Interestingly enough, the value of a string constant is not the text, but a pointer to the first character of the text. Thus when you write

```
cout << "Please type your name:";
```

you are merely using the general form mentioned in Section 13.1.1. You may also write

```
char *name;
name = "Einstein";
```

Even in this statement, the right hand side of the assignment, `"Einstein"` denotes the address in memory of where the text `"Einstein"` is stored (terminated by a '\0' as always). Thus it is fine to store this in a variable of type `char*`. Of course, if you subsequently write `cout << name;`, you will see `"Einstein"` printed.

13.1.4 Examples

Character arrays behave like ordinary integer arrays, except when it comes to reading and printing, and in that they contain a '\0' character which marks the end of the useful portion of the array. So processing them is reasonably straight forward. Note that characters are a subtype of integers, and as such we can perform arithmetic on characters, and compare them, just as we do for integers.

Our first example is a function for determining the length of the text stored in a char array.

```
int length(char *txt){
  int L=0;
  while(char[L] != '\0') L++;
  return L;
}
```

The function takes a single argument, say the array name (or the pointer to the zeroth element of the array). Notice that the actual length of the array is not needed. This is because we access elements only till the null character. Indeed, the function simply steps through the elements of the array, and returns the index at which it finds the null character, '\0'. Since the starting index is 0, the null character will be at index equal to the length of the text string.

Our second example is a function for copying a string stored in an array `source` to another array `destination`. This is like copying other arrays, except that we must only worry about the useful portion of the source array, i.e. till the occurrence of the '\0' character. The function does not worry at all about the lengths of the 2 arrays as defined, it is assumed that the call has been made ensuring that indices will not exceed the array bounds.

```
void strcpy(char destination[], char source[])
// precondition: '\0' must occur in source.  destination must be long
// enough to hold the entire string + '\0'.
{
  int i;
  for(i=0; source[i] != '\0'; i++)
    destination[i]=source[i];
  destination[i]=source[i];    // copy the '\0' itself
}
```

As an example of using this, note that a string constant can be used any place a pointer to char is needed. Thus we can write `strcpy(name,"Einstein")` which would simply set the name to "Einstein".

Here is a more interesting function: it takes two strings and returns which one is lexicographically smaller, i.e. would appear first in the dictionary. The function simply compares corresponding characters of the two strings, starting at the 0th. If the end of the strings is reached without finding unequal characters, then it means that the two strings are identical, in which case we must return `'='`. If at some comparison we find the character in one string to be smaller than the other, that string is declared smaller. If one string ends earlier, while the preceding characters are the same, then the string that ends is smaller.

This logic is implemented in the code below. We maintain the loop invariant: at the beginning of the loop characters 0 through i-1 of both arrays must be non null and identical. So if we find both `a[i]` and `b[i]` to be null, clearly the strings are identical and hence we return 0. If `a[i]` is null but not `b[i]`, then `a` is a prefix of `b`. Because prefixes appear before longer strings in the dictionary, we return `'<'`. We proceed similarly if `b[i]` is null but not `a[i]`. If `a[i]>b[i]` we return `'>'`, if `a[i]<b[i]` we return `'<'`. If none of these conditions apply, then the `i`th character in both strings must be non-null and identical. So the invariant for the next iteration is satisfied. So we increment `i` and go to the next iteration.

```
char compare(char a[], char b[])
// returns '<' if a is smaller, '=' if equal, '>' if b is smaller.
{
  int i = 0;
  while(true){                  // Invariant: a[0..i-1] == b[0..i-1]
    if(a[i] == '\0' && b[i] == '\0') return '=';
    if(a[i] == '\0') return '<';
    if(b[i] == '\0') return '>';
    if(a[i]<b[i]) return '<';
    if(a[i]>b[i]) return '>';
    i++;
  }
}
```

This may be called using the following main program.

```
main(){
  char a[40], b[40];
  cin.getline(a,40);
  cin.getline(b,40);
  cout << a << " " << compare(a,b) << " " << b << endl;
}
```

If you execute this program, it would expect you to type two lines. Say you typed:

```
Mathematics
Biology
```

then it would print out > and stop, because “Mathematics” appears after “Biology” in the dictionary order.

13.2 Two dimensional Arrays

Sequences of numbers are naturally represented as arrays. However, we will run into objects like matrices which are collections of elements described using two indices. For such cases, C++ provides 2 dimensional arrays.

Here is an example of how a two dimensional array might be defined:

```
double a[m][n];  // m,n must be compile time constants as per C++ standard.
                // But s++ will allow integer expressions.
```

This causes space for m*n doubleing point variables to be allocated. These variables are accessed as a[i][j] where we require $0 \leq$ i $<$ m, and $0 \leq$ j $<$ n. The variables are stored in the so called row major order in memory, i.e. in the order a[0][0], a[0][1], ... a[0][n-1], a[1][0], ... a[1][n-1], ... a[m-1][n-1]. The numbers m,n are said to be the first and second dimension of the array. We will also refer to them as the number of rows and the number of columns respectively.

Manipulating 2 dimensional arrays is similar to 1 dimensional – we will just have 2 loops over the two indices rather than just one.

Here is a program fragment that reads in two matrices and prints their product. Remember that if A is an $m \times n$ matrix, and B an $n \times p$ matrix, then there product is an $m \times p$ matrix C where

$$c_{ij} = \sum_{k=1}^{n} a_{ik} \cdot b_{kj}$$

where we have let the array indices start at 1, as is customary in Mathematics. The code below, of course, starts indices at 0. The code also shows how a two dimensional array can be initialized in the definition itself if you wish. The values for each row must appear in braces, and these in turn in an outer pair of braces.

```
double a[3][2]={{1,2},{3,4},{5,6}}, b[2][4]={{1,2,3,4},{5,6,7,8}}, c[3][4];

for(int i=0; i<3; i++)
   for(int j=0; j<4; j++){
      c[i][j] = 0;
      for(int k=0; k<2; k++)
         c[i][j] += a[i][k]*b[k][j];
   }

for(int i=0; i<3; i++){
   for(int j=0; j<4; j++) cout << c[i][j] << " ";
   cout << endl;
}
```

We may define two dimensional arrays of chars, with initialization, which is of course optional. For example we could write:

```
  char countries[6][20] = {"India","China","Sri Lanka","Nepal",
                           "Bangladesh","Pakistan"};
```

Here the first string, "India" is deemed to initialize the zeroth row, and so on for the six strings.

Applying only one index to the name of a two dimensional array returns the address of the zeroth element of the corresponding row. For character arrays, this is the way to refer to one of the strings stored. Thus countries[i] will return the address of the zeroth character of the ith string stored in the array, in other words, the address of the ith string. So if we write compare(countries[0], countries[1]), where compare is as defined in Section 13.1.4, it would return '<' as the result because India will precede Sri Lanka in the dictionary order.

Here is a program which has two arrays, countries which lists countries, and capitals which lists corresponding capitals. It takes as input a string from the keyboard. It prints out the name of the corresponding capital if the string is in the list of countries stored in countries. This check is made using our compare function. Note that the function must return '=' if the two arguments are identical strings.

```
int main(){
  const int wordLength = 20;
  char countries[6][wordLength] = {"India","China","Sri Lanka","Nepal",
                           "Bangladesh","Pakistan"};
  char capitals[6][wordLength] = {"New Delhi","Beijing","Colombo","Kathmandu",
                           "Dhaka","Islamabad"};
  char country[wordLength];
  cout << "Country: ";
  cin.getline(country,wordLength);

  int i;
  for(i=0; i<6; i++){
    if(compare(country,countries[i]) == '='){
      cout << capitals[i] << endl;
      break;
    }
  }
  if(i == 6) cout << "Dont know the capital.\n";
}
```

When the loop terminates, we know that `i` must be strictly less than 6 if the country was found in `countries`, and equal to 6 if not found. Hence we print the message that we dont know the capital only if `i` is 6 at the end.

13.2.1 Passing 2 dimensional arrays to functions

It is possible to pass a two dimensional array to a function. However, in the called function, the second dimension of the array parameter must be given as a compile time constant. Thus we might write:

```
void print(char countries[][20], int noOfCountries){
  for(int i=0; i<noOfCountries; i++) cout << countries[i] << endl;
}
```

This may be called as `print(countries,6)`, where the second argument is the first dimension of the `countries` array. It will print out the countries on separate lines.

This is not too useful, because any such function can only be used for arrays in which the second dimension is 20. For example, this makes it impossible to write a general matrix multiplication function for matrices of arbitrary sizes. So in the next section we will see a two dimensional array that has been provided as a part of `simplecpp`. This can be passed to functions, and the functions can be written without having to know at compile time either the first or the second dimension of the array!

But if we do know the second dimension, then the standard two dimensional arrays are useful. Here is how they can be used in drawing polygons in `simplecpp` graphics.

13.2.2 Drawing polygons in `simplecpp`

`simplecpp` contains the following command for drawing polygons:

```
Polygon pName(cx,cy,Vertices,n);
```

This will create a polygon named `pName`. The parameters `cx,cy` give the rotation center of the polygon. The parameter `n` is an integer giving the number of vertices, and `Vertices` is a two dimensional `double` array with `n` rows and 2 columns, where each row gives the x,y coordinates of the vertices, relative to the center `(cx,cy)`. A polygon is a shape in the sense of Chapter 4, so we may use all the commands for shapes on polygons.

The boundary of the polygon is traced starting at vertex 0, then going to vertex 1 and so on till vertex `n-1`. Note that the boundary may intersect itself.

Here is an example. We create a regular pentagon and a pentagonal star. Then we rotate them.

```
main_program{
  initCanvas("Pentagon");
  double pentaV[5][2], starV[5][2];

  for(int i=0; i<5; i++){
    pentaV[i][0] = 100 * cos(2*PI/5*i);
    pentaV[i][1] = 100 * sin(2*PI/5*i);
    starV[i][0] = 100 * cos(4*PI/5*i);
    starV[i][1] = 100 * sin(4*PI/5*i);
  }

  Polygon penta(200,200,pentaV,5);
  Polygon star(200,400,starV,5);

  for(int i=0; i<100; i++){
    penta.left(5);
    star.right(5);
    wait(0.1);
  }

  getClick();
}
```

Note that there is a more natural ways of specifying the star shape: consider it to be a (concave) polygon of 10 vertices. Thus we could have given the coordinates of the 10 vertices in order. Calculating the coordinates of the "inner" vertices is a bit messy, though.

13.3 Arrays of Pointers

An array is really a sequence in memory of variables of the same type. We have seen arrays of `int`, `double`, `char`, but we can have arrays of any type of variable. So you might ask, can we have arrays of pointers? It is certainly possible, and it turns out to be useful too.

We can create an array of 10 variables, each of type pointer to `int` by writing the following.

```
int *y[10];
```

This statement is undoubtedly confusing. The way to understand it is to compare it with a usual array definition.

```
int x[10];
```

You can read this statement as saying "`x[i]` is an `int` for `i=0` to `i=9`." In a similar manner, you should read the statement `int *y[10];` as saying "`*y[i]` is an `int` for `i=0` to `i=9`." But if content of `y[i]` is an `int`, then `y[i]` must be an `int` pointer.

Once you have defined an array of pointers, you can store addresses of appropriate variables in each element of the array. For example, you might write something like:

```
int *y[10];
int z = 100;
y[0] = &z;
cout << *y[0] << endl;
```

This will print 100, because `y[0]` contains the address of `z`, and hence `*y[0]` just means `z`, and hence the value of `z`, 100, will be printed. This use of arrays of pointers is not very interesting.

However, arrays of pointers to `char` can be used very nicely, as the following code fragment shows.

```
char *weekdays[7];
weekdays[0] = "Monday";
weekdays[1] = "Tuesday";
weekdays[2] = "Wednesday";
weekdays[3] = "Thursday";
weekdays[4] = "Friday";
weekdays[5] = "Saturday";
weekdays[6] = "Sunday";
```

The first statement defines `weekdays` to be an array of pointers to `char`. The subsequent statements initialize each of the elements of the array. Each element is of type `char*`, and we pointed out in Section 13.1.3 that quoted text such as `"January"` really represents the address in memory where the string `"January"` is stored. Thus `weekday[i]` will be set to point to the position in memory where the name of the `i`th day of the week is stored. Hence if we write

```
cout << weekdays[4];
```

we will be printing `"Friday"`. Note that we could have defined `weekdays` differently:

```
char weekdays[7][10] = {"Monday", "Tuesday", "Wednesday", "Thursday",
                         "Friday", "Saturday", "Sunday"};
```

Even with this definition we would write `weekdays[4]` to mean `"Friday"`. However, note that in this case we have used 10 bytes (length of the longest name, plus one byte to store '\0') to store each name. The first definition however uses exactly the number of bytes needed for each name, plus one byte to store the `'\0'`.

13.3.1 Command line arguments to `main`

So far, we have executed C++ programs by specifying the name of the executable file, usually `a.out`, on the command line. Specifically, the program is executed by typing:

```
a.out
```

or `./a.out` on the shell command line. This causes the `main` function in your program to be called.

But you may execute your program differently. C++ does allow you to provide additional text after `a.out`, and this text can be processed by your program. For example, you may write:

```
a.out Mathematics Biology
```

In this case your program can be told that you have typed the words *Mathematics* and `Biology` after `a.out`. This can be done using an alternative (overloaded) declaration provided for `main`.

```
int main(int argc, char *argv[]);
```

Thus `main` may take two arguments. The first is an integer argument `argc`. The second argument is an array (since it ends in `[]`) has name `argv`, and each element is of type `char*`. In other words, `argv` is an array of pointers to `char`.

Suppose you use this form of `main`. Then when you execute your program, the Operating System calls the function `main`, but also passes some parameters. Specifically the following are the values passed in the parameters:

1. `argc` gives the number of words typed on the command line, including the name of the executable program (`a.out` or other).

2. The argument `argv` is an array of `argc` elements, with the `i`th element `argv[i]` being the address of the `i`th command line word (typically called `i`th command line argument).

Thus if you had invoked the main program by writing `a.out Mathematics Biology` the value of `argc` would be 3. The parameter `argv` would have 3 elements of type `char*`, and these would respectively be addresses of the text strings (null terminated) `"a.out"`, `"Mathematics"`, and `"Biology"` respectively.

Here is a simple program that just prints out the values of all its command line arguments.

```
int main(int argc, char *argv[]){
  for(int i=0; i<argc; i++) cout << argv[i] << endl;
}
```

This program when invoked as `a.out Mathematics Biology` would print out

```
a.out
Mathematics
Biology
```

Of course, you can do more interesting processing on the command line arguments. See Appendix H.

13.4 A home-made 2 dimensional array

In `simplecpp` we have provided an alternate mechanism to represent two dimensional data. You may write, for example:

```
matrix<double> abc(3,5);
```

This will *essentially* give you a 2 dimensional array of `doubles`. The array is named `abc`, and this can be any identifies as usual. The dimensions of the array are 3, 5, and these are specified not in two pairs of square brackets, but in a single pair of parentheses, separated by a comma. Indexing is also to be done using a similar notation, e.g. `abc(i,j)` will refer to the element in row `i` and column `j`. The name `abc` has type `matrix<double>`. Note that by using other types instead of `double` inside the angled brackets `<>` following `matrix` you can get 2 dimensional arrays of other types too. In what follows, we will write `matrix` to mean `matrix<T>` for all possible types `T`.

There are several differences between standard C++ two dimensional arrays and this new 2 dimensional array.

1. You can get the number of rows and columns in a `matrix` such as `abc` by writing respectively `abc.rows()`, and `abc.columns()`. These would respectively evaluate to 3,5 for the definition of `abc` as above.

2. A `matrix` can be conveniently passed to functions. The name must be passed, and nothing else is needed. If the name of the corresponding parameter is `pqr`, then we can get the rows and columns in the passed matrix by writing `pqr.rows()` and `pqr.columns()`.

3. Data of type `matrix` is passed by value, i.e. a new copy is made for the called function. Thus if you want to modify a parameter, then it should be marked a reference parameter. Usually matrices will be large, and hence it would be good to avoid copying. So it is a good idea to pass all matrices by reference. Those matrices that will not be modified by the function should be marked `const`.

4. Whenever you access an element i.e. write `abc(i,j)` it is first checked whether the indices `i,j` are in the required range. If they are not, then a message is printed, giving the allowed range, and the indices that were actually used. If the indices are in the range, then you get access as usual.

5. A `matrix` object cannot be initialized as a part of the definition. You have to explicitly write assignment statements to do so.

6. You can write `cout << abc;` which will cause the array to be printed in the row major order, one row per line.

7. You can write `cin >> abc;` which will cause array elements to be read from the keyboard in row major order.

In standard two dimensional arrays, if we supply only one index, we get the starting address of the corresponding row. This feature continues to hold. We can indeed write `abc(i)` to get the starting address of the `i`th row.

We give examples which illustrate these features.

13.4.1 Matrix multiplication function

Suppose we wish to multiply matrices represented by arrays `a,b` and produce a matrix `c`. Here is the function for doing it.

```
void matmult(matrix<double> & c, matrix<double>& a, matrix<double> & b)
  // precondition: number of rows in c = number of rows in a,
  //   number of columns in a = number of rows in b
  //   number of columns in b = number of columns in c
  //
  // c should be different from a,b.
{
  for(int i=0; i<c.rows(); i++)
    for(int j=0; j<c.columns(); j++){
      c(i,j) = 0;
      for(int k=0; k<a.columns(); k++)
        c(i,j) += a(i,k)*b(k,j);
    }
}
```

We want the result back in `c`, so that is a reference parameter. The other parameters are `const` reference As you can see, the body is essentially the same as in Section 13.2, except that we have picked up the loop bounds from the rows/columns of the argument matrices.

A possible main program is as follows.

```
main(){
  matrix<double> a(2,3), b(3,1), c(2,1);

  cin >> a;
  cin >> b;

  matmult(c,a,b);

  cout << c;
}
```

We consider one more example.

```
main(){
  matrix<char> names(2,20);

  strcpy(names(0),"Anita"); // names(0) = address where 0th string starts.
  strcpy(names(1),"Raju");

  cout << names(0)<< endl;
  cout << names(1)<< endl;

  names(0,1)='m';
  cout << names(0)<< endl;
}
```

The expression `names(0)` gives the address where the zeroth string can begin, and indeed we can use the `strcpy` function defined in Section 13.1.4 to store data into it. Similarly we store into `names(1)`.

The last two lines show the effect of changing one element. You should see "Amita" get printed rather than "Anita".

13.5 Linear simultaneous equations

One of the most important uses of matrices and two dimensional arrays is to represent linear simultaneous equations. Say we are given simultaneous equations:

$$\begin{aligned} 3x_2 + 5x_3 &= 10 \\ 2x_1 + 6x_2 + 8x_3 &= 38 \\ 7x_1 + 4x_2 + 9x_3 &= 22 \end{aligned}$$

Then they can be conveniently represented by the matrix equation

$$\begin{bmatrix} 0 & 3 & 5 \\ 2 & 6 & 8 \\ 7 & 4 & 9 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} 10 \\ 38 \\ 22 \end{bmatrix}$$

Denoting the matrix by A, the vector of unknowns by x and the right hand side vector by b, we have the matrix equation $Ax = b$ in which we are to solve for x given A, b.

The direct way to solve a system of equations is by a process called *Gaussian elimination*[1], in fact a form of it called Gauss-Jordan elimination.

Observe first that if the matrix A was the identity matrix, i.e. $a_{ii} = 1$ and $a_{ij} = 0$ for all $i, j \neq i$, then the problem is very easy. Multiplying out we would get $x = b$. Thus for this b is itself the solution. This suggests a strategy. We will make modifications to A, b such that the modifications do not change the solution of $Ax = b$. If at the end of the sequence of modifications, our matrix A becomes the identity matrix then the value of b at that time would itself be the solution.

It turns out that several operations performed on the system of equations (and hence A, b) indeed do not change the solutions to the system. One such operation is to multiply any equation by a constant. This is akin to multiplying a row of the matrix A and the corresponding element of the vector b by a (the same) constant. Another operation is to add one equation to another, and replace the latter equation by the result. In our example, say we add the first equation to the second. Thus we get the equation $2x_1 + 9x_2 + 13x_3 = 48$. We replace the second equation with this equation. This is succinctly done in the matrix representation: we merely add the first row of A to the second row, and the first element of b to the second element of b. Thus the second row of A would then become [2 9 13] and the second element of b would become 48, while the other elements remained the same.

[1] The method is actually much older than Gauss.

We now show how we can change A, b, without changing the solution, so that the first column of A becomes 1,0,0 (read top to bottom), i.e. identical to the first column of the identity matrix. The same process can then be adapted for the other columns.

1. If the coefficient of x_1 is zero in the first equation, pick any equation which has a non zero coefficient for x_1. Suppose the ith equation has a non-zero coefficient for x_1. Then exchange equation 1 and equation i. This corresponds to exchanging row 1 and row i of A and also element 1 and element i of b. Doing this for our example we get:

$$\begin{bmatrix} 2 & 6 & 8 \\ 0 & 3 & 5 \\ 7 & 4 & 9 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} 38 \\ 10 \\ 22 \end{bmatrix}$$

2. Divide the first equation by the coefficient of a_{11}. We thus get

$$\begin{bmatrix} 1 & 3 & 4 \\ 0 & 3 & 5 \\ 7 & 4 & 9 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} 19 \\ 10 \\ 22 \end{bmatrix}$$

3. For each i, add $-a_{i1}$ times the first equation to equation i. Say we do this for row 2. Thus we must add $-a_{21} = 0$ times the first row. So nothing need be done. So we then consider row 3. Since $a_{31} = 7$, we add -7 times the first equation to equation 3. Thus we now have:

$$\begin{bmatrix} 1 & 3 & 4 \\ 0 & 3 & 5 \\ 0 & -17 & -19 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} 19 \\ 10 \\ -111 \end{bmatrix}$$

It should be clear that the above process would indeed make the first column identical to the first column of the identity matrix. In a similar manner, you should be able to get the other columns to match the identity matrix.

The first step in the above description deserves more explanation. Suppose you have managed to make the first $j-1$ columns of A resemble the first $j-1$ columns of the identity matrix. Now the first step above instructs you to find an equation in which the coefficient of x_j is non-zero. For this, you should only look at equations j through n, and not consider the first $j-1$ equations. This step may or may not succeed. It will not succeed if $a_{kj} = 0$ for all $k = j \ldots n$. In this case, it turns out that the system of equations does not have a unique solution; it may have many solutions or no solutions at all. In this case you should report failure.

The code for doing all this is left as an exercise. You are expected to write a function which does this, having the following prototype

```
bool solveLS(matrix<double> & A, double b[], double x[]);
// A must be an n x n matrix, and b, x must be n element vectors.
// The solution to Ax=b will be computed and returned in x if it is unique.
// If the solution is unique, true will be returned as the result.
// If the solution is not unique, then x will not be altered, and
// false will be returned.
```

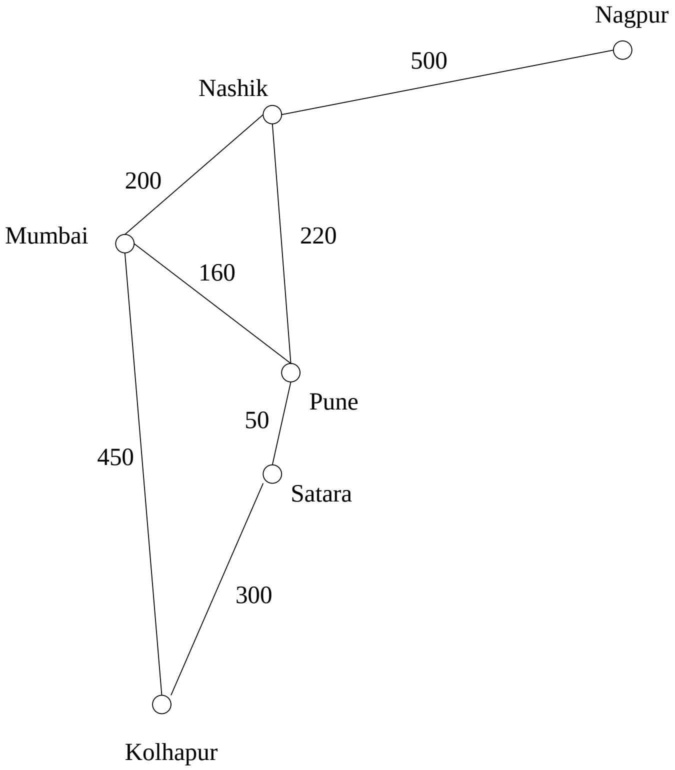


Figure 13.1: Schematic Map

13.6 All source shortest paths

Suppose we have a road network, as in Figure 13.1. As you can see there are points representing towns and lines connecting them representing roads. Next to each road is given the length of the road in km. Our goal is to take this information, and print out the length of the shortest path between every pair of towns. Sometimes we do not need the shortest path length from every town ("all source") to every other town, but only from a specific town to other towns. We will consider such "single source" problems in Section 21.4.

Note that in our terminology of Section 10.2 the road network is a *graph* in which towns are *vertices* and roads are *edges.* Graphs in which edges have numbers associated with them are said to be *weighted*, with the number called the weight. In these terms, our graph is weighted, with the length of the road being the weight of the corresponding edge. In our problem, the numbers represent the length of the roads, and so we will call them edge lengths rather than edge weights.

The first question, of course, is how to represent our graph, which we will call G. A matrix turns out to be rather convenient for this. If there are n towns/vertices, we will use an $n \times n$ matrix, say D. We will call this the direct connection matrix, because it will store information about direct road connections. In general, the length (or weight) of the edge from i to j, which in our case is the length of the road from town i to town j is stored in d_{ij}. Note that this leaves open the possibility of storing different values in d_{ij} and d_{ji}. This is relevant in our case: the distance in one direction can be different than the other, especially in hilly areas. However, you may assume for simplicity that $d_{ij} = d_{ij}$. These numbers must be given to us as part of the input.

It needs some intuition to decide what to store in d_{ij} if there is no road from town i to $j \neq i$. The correct number to store turns out to be ∞. The intuition behind this is: a road of length ∞ can be considered useless for travelling, and hence equivalent to no road. We will shortly see how to represent ∞ in C++. The only entries of D we have not defined so far are the diagonal entries, d_{ii}. We set all $d_{ii} = 0$. The intuition here is that the direct distance from a town to itself should be considered 0.

It turns out that in C++ you can indeed represent ∞. There is a special bit pattern for representing ∞ in the IEEE doubleing point standard (Appendix E) which C++ implements, and the name given to the bit pattern is `HUGE_VAL`. So indeed you may write:

```
double distance = HUGE_VAL;
```

The convenience of this notation is that `HUGE_VAL` behaves like infinity! If you multiply a number and infinity you expect to get infinity, and indeed this happens. If you find the minimum of some number and infinity you expect to get that number. Or if you divide any finite number by infinity, you expect that the result will be zero. All such expectations are satisfied! Without you doing anything special!

For the data in our map, our matrix would be (with 1=Mumbai, 2=Pune, 3=Nashik, 4=Kolhapur, 5=Nagpur, 6=Satara):

$$D = \begin{bmatrix} 0 & 160 & 200 & 450 & \infty & \infty \\ 160 & 0 & 220 & \infty & \infty & 50 \\ 200 & 220 & 0 & \infty & 500 & \infty \\ 450 & \infty & \infty & 0 & \infty & 300 \\ \infty & \infty & 500 & \infty & 0 & \infty \\ \infty & 50 & \infty & 300 & \infty & 0 \end{bmatrix}$$

13.6.1 Algorithm

Let us first define an unusual matrix multiplication. Suppose P, Q are $m \times n$ and $n \times p$ matrices respectively. Then $R = P \otimes Q$ is defined as the $m \times p$ matrix in which entry $r_{ij} = \min(p_{i1} + q_{1j}, p_{i2} + q_{2j}, \ldots, p_{in} + q_{nj})$. Notice that this is same as ordinary matrix multiplication, except that we perform the min operation instead of addition, and addition instead of multiplication.

Our algorithm for calculating shortest paths is extremely simple. It requires us to compute $X = D^{n-1}$, where matrix multiplication is performed using the operator $\otimes$. Then x_{ij} gives the length of the shortest path from i to j. Why this is so will be explained in Section 13.6.3. Note that x_{ij} might turn out to be ∞. In this case, the interpretation is that there is no path at all that goes from i to j.

You might think that to compute D^{n-1} we will need to perform $n-2$ matrix multiplications. However, this is not true. We simply square D repeatedly, specifically we use the following algorithm.

```
1. $X = D$

2. For $t = 1$ to $p = \lceil \log_2(n-1) \rceil$ do
```

```
        Compute $X = X \otimes X$.
3. end for
4. return $X$.
```

After one iteration of the above algorithm we will have $X = D^2$, after two we will have $X = D^4$, then $X = D^8$ and so on, and after $p = \lceil \log_2(n-1) \rceil$ we will have $X = D^{2^p} = D^{n'}$ where $n' = 2^p$ is the smallest power of 2 no smaller than $n-1$. We will see shortly that $D^{n'} = D^{n-1}$. Note that using the algorithm above, we have calculated $D^{n'}$ using $\lceil \log_2(n-1) \rceil$ matrix multiplications, much less than the obvious $n-2$.

13.6.2 Execution example

Let us execute one iteration of the above algorithm for our matrix D as defined earlier. Thus we will compute $X = D^2 = D \otimes D$. Thus we will have

$$x_{ij} = \min\{d_{ik} + d_{kj} \mid k = 1, 2, \ldots, n\}$$

Thus the entry x_{ij} for $i = 3$ (Nashik) and $j = 4$ (Kolhapur) will be

$$\begin{aligned} x_{34} &= \min\{d_{31} + d_{14},\ d_{32} + d_{24},\ d_{33} + d_{34},\ d_{34} + d_{44},\ d_{35} + d_{54},\ d_{36} + d_{64}\} \\ &= \min\{200 + 450,\ 220 + \infty,\ 0 + \infty,\ \infty + 0,\ 500 + \infty,\ \infty + 300\} \\ &= 650 \end{aligned}$$

Note that 650 is *not* the length of the shortest path from Nashik to Kolhapur, however it is a good estimate of the actual shortest path length (570, going through Pune and Satara). In our matrix D, we had $d_{34} = \infty$, i.e. there was not direct connection. So as we go to D^2, our estimate has improved considerably, from ∞ to 650, though not perfectly. On the other hand, if you work it out you will see that $x_{16} = 210$, which is indeed the shortest path length between Mumbai and Satara. On the other hand, x_{54} and d_{54} are both ∞, and so there is no progress on this.

Thus there is some progress as we go to D^2 from D. Indeed, when we are done computing $D^{n'}$, we will have all correct lengths, as we will argue next.

13.6.3 Explanation of the algorithm

For the ease of discussion it is convenient to consider two additional kinds of edges into our graph. The first are the so called *self-loop* edges going from every vertex i to itself. We associate the length d_{ii} which we already set to 0, with such edges. Further, we will consider a *ghost* edge from vertex i to vertex $j \neq i$ if no real edge exists. A ghost edge will be associated with the length d_{ij} which we already set to ∞ earlier. In the first part of our analysis we will consider *general* paths in which all 3 kinds of edges appear, real, self-loop and ghost. Of course the edges in a path must be continuous, i.e. the mth edge must begin in the vertex in which the $m-1$th edge ends. In what follows *path* will denote a genaral path unless mentioned otherwise.

We will use $L(p)$ to denote the length of a (general) path p, i.e. the sum of the lengths of the edges in it. Obviously, $L(p) = \infty$ if p contains even one ghost edge. We will use $E(p)$ to denote the number of edges in path p.

The main idea of the algorithm is in the following theorem.

Theorem 3 *Let $X = D^m$ for any integer m. Then x_{ij} gives the length of a shortest path from i to j having exactly m (real/self-loop/ghost) edges, i.e.*

$$x_{ij} = \min\{L(p) \mid p \textit{ is a path from } i \textit{ to } j \textit{ and } E(p) = m\ \}$$

Let us first observe that the theorem is true for the case $m = 1$. Indeed we defined D so that d_{ij} denoted the length of the real/self-loop/ghost edge from i to j.

Let us now consider the theorem for $m = 2$. In this case we have $X = D^2$. The theorem now says that x_{ij} must be the length of a shortest path from i to j having 2 edges. Before we prove this, let us check if this worked out correctly in our example. In the example, we first saw that $x_{34} = 650$. This is indeed the shortest path from Nashik to Kolhapur having 2 edges – the path passing through Mumbai. The actual shortest path which goes through Pune and Satara has 3 edges, and so is not to be considered. Likewise, we saw that $x_{16} = 210$ which is indeed the length of the shortest path having at most 2 edges. Finally, $x_{54} = \infty$, and indeed all paths between Nagpur and Kolhapur having 2 edges must contain some ghost edge, and hence the shortest length is ∞.

Now we prove the theorem for $m = 2$. When $X = D^2$ we have

$$x_{ij} = \min\{d_{ik} + d_{kj} \mid k = 1, 2, \ldots, n\}$$

The key observation is: each sum $d_{ik} + d_{kj}$ in the right hand side is the length of a 2 edge path from i to j. In fact, you will see that all possible paths are considered, since we consider all choices for k. Thus x_{ij} gives the length of a shortest 2 edge path, proving the claim.

We now sketch the idea of how the proof can be extended for larger m. Suppose now that we have a graph G' corresponding to our matrix $X = D^2$, i.e. G' contains an edge from each i to each j of length x_{ij}. What if we now compute X^2? We would get lengths of shortest 2 edge paths in G'. However, note that edges of G' are 2 edge paths of our original graph G. Hence shortest 2 edge paths of G' are shortest 4 edge paths of G. But $X^2 = D^4$, and hence the entries of D^4 give the lengths of the shortest 4 edge paths of G. We can keep repeating this argument to get for larger m.

The final question is, how does this relate to what we want: lengths of shortest paths with only real edges, and doesn't matter how many such edges there are. This is what we consider next.

Lemma 1 *Suppose $X = D^{n'}$ denotes the matrix returned by the algorithm. Suppose there exist real paths from a vertex i to a vertex j in G. Suppose P has the shortest length amongst these. Then $x_{ij} = L(P)$.*

Proof: By Theorem 3, with $m = n'$, we know x_{ij} is the length of the shortest path from the set

$$S = \{p \mid p \text{ is a path from } i \text{ to } j \text{ and } E(p) = n'\}$$

i.e. $x_{ij} = \min_{p \in S} L(p)$. On the other hand we want the length of the shortest path from the paths in the set

$$\tilde{S} = \{p \mid p \text{ is a } \textit{real} \text{ path from } i \text{ to } j, \text{ with no bound on the number of edges}\}$$

We will argue that the minimum over $\tilde{S}$ and the minimum over S are identical.

Suppose P is a path from i to j with $E(P) < n'$. Now consider P' obtained by adding $n' - E(P)$ self-loops from i to itself at the beginning of P. P' is now a path from i to j, with $E(P') = E(P)$. Further, $L(P') = L(P)$, since self-loops have length 0. But now P' belongs to the set S. So $L(P')$ cannot be smaller than the shortest length x_{ij} in S, i.e. So $x_{ij} \leq L(P') = L(P)$. In other words, if we added P into the set S, its minimum would not change. But this argument applies to all possible paths P, i.e. any path with length less than n'. So defining

$$S' = \{p \mid p \text{ is a path from } i \text{ to } j \text{ and } E(p) \leq n' \}$$

We get $\min_{p \in S} L(p) = \min_{p \in S'} L(p)$.

We next turn to $\tilde{S}$. Consider a shortest path $\tilde{P}$ in it. Since $\tilde{P}$ is shortest, it cannot pass through any vertex twice. But we only have n vertices. Thus, $\tilde{P}$ can have at most n vertices, and hence $n-1$ edges, i.e. $E(\tilde{P}) \leq n-1$. Further, we chose $n' \geq n-1$. Thus, $E(\tilde{P}) \leq n'$. Consider the set

$$\tilde{S'} = \{p \mid p \text{ is a } \textit{real} \text{ path from } i \text{ to } j \text{ and } E(p) \leq n' \}$$

Clearly, the $\tilde{S'} \subseteq \tilde{S}$, and a shortest path $\tilde{P} \in \tilde{S}$ is also in $\tilde{S'}$. Hence we have

$$\min_{p \in \tilde{S}} L(p) = L(\tilde{P}) = \min_{p \in \tilde{S'}} L(p)$$

Suppose we now allow the paths in $\tilde{S'}$ to have self-loops. This would increase the number of possible paths in $\tilde{S'}$, but the shortest path would not change, and hence the length of the shortest path would not change either. We could also allow some edgess to be ghost edges. All such paths with ghost edges will have infinite length, and hence they will also not change the minimum. But when we allow these changes, the new set we get is simply S', which we defined above! We have proved that both sets must have the same minimum, i.e.

$$\min_{p \in S'} L(p) = \min_{p \in \tilde{S'}} L(p)$$

Thus we have proved by our sequence of deductions that $x_{ij} = \min_{p \in S} L(p) = \min_{p \in S'} L(p) = \min_{p \in \tilde{S'}} L(p) = \min_{p \in \tilde{S}} L(p)$ But the last quantity is in fact the value desired. ■

Alternate Proof: We first characterize P. Since P is shortest, it will not return to any vertex it visited earlier, i.e. all the vertices on it must be distinct. But there are only n vertices overall, and a path with n vertices will only have $n-1$ edges. Thus $E(P) \leq n-1$. But we chose n' to be the smallest power of 2 such that $n' \geq n-1$. Thus we have $E(P) \leq n'$. Suppose we add $n' - E(p)$ copies of the self loop from i to itself at the beginning of P. What we get is a path P' that also goes from i to j, but which has exactly n' edges. Further, $L(P') = L(P)$, because the edges we added had length 0. By Theorem 3, we have

$$x_{ij} = \min\{L(p) \mid p \text{ is a path from } i \text{ to } j \text{ and } E(p) = n' \}$$

where we know that x_{ij} is the length of some path $\tilde{P}$. Noting that the set over which the minimum is taken includes P', we have $L(\tilde{P}) = x_{ij} \leq L(P')$. But $L(P') = L(P)$ and P has finite length. Thus $\tilde{P}$ has finite length, and hence it cannot contain any ghost edges in it. It may contain self-loop edges, which we drop and get a real path $\tilde{P}'$, for which we must have $L(\tilde{P}') = L(\tilde{P})$, since we only dropped length 0 edges. Because P is a shortest real path from i to j, we must have $L(P) \leq L(\tilde{P}')$. Putting together everything we get

$$L(P) \leq L(\tilde{P}') = L(\tilde{P}) = x_{ij} \leq L(P') = L(P)$$

Thus all the terms must be equal. Thus $L(P) = x_{ij}$.

Lemma 2 *Suppose $X = D^{n'}$ denotes the matrix returned by the algorithm. Suppose there is no real path from a vertex i to a vertex j in G. Then $x_{ij} = \infty$.*

Since there is no real path from i to j, there cannot be a path from i to j having only real and self-loop edges. Thus, every path from i to j must include at least one ghost edge. Thus its length is infinite. Since this is true for all paths from i to j, a shortest among them must also have infinite length.

13.7 Generating permutations

We consider the problem of printing out all permutations of the integers 0 to $n-1$. As you know, there are $n!$ permutations of n objects, we would like all these printed. As an example, if $n = 3$, we would like the output to be something like:

```
0 1 2
0 2 1
1 0 2
1 2 0
2 0 1
2 1 0
```

If you try to solve this problem by hand on paper, you will realize that it helps to think of some systematic order in which to generate the permutations. For example, we could consider generating all permutations starting with 0, then all starting with 1, and so on. How do we generate all permutations starting with 0? Presumably we apply the same idea again (recurse!): we first generate all permutations starting with 01, then those that start with 02 and so on.

We will write a recursive function `gPerm` to print all permutations of a given set. In the intermediate stages, our problem will be slightly different; we dont want to print all permutations, but all permutations given some starting portion of the permutation. So it will be convenient if our function takes two arguments: a sequence P which has been fixed as the starting portion, and a set R consisting of the remaining elements which will make the remaining part of the permutation. In this notation, if we want to print all permutations of the set $\{0,\ldots,n-1\}$ that start with 02 we call our function with $P = 02$, and $R = \{1,3,4,\ldots,n-1\}$. If we merely want all permutations of $R = \{0,\ldots,n-1\}$, then we can call our function with P being the empty string and $R = \{0,\ldots,n-1\}$.

We have already hinted how `gPerm(P,R)` will recurse. The fixed part `P` will grow, and it can grow by using any element of `R`. Thus the recursive portion of `gPerm(S,R)` can be written as follows.

```
For each r in R begin
  (a) P' = P with r appended.
  (b) R' = R - r, where subtraction is used to denote removing from the set.
  (c) Call gPerm(P',R').
end
```

We have already given examples of this for the cases `P = ""`, and `P = "0"`.

The base case for the recursion will arise when `R` is the empty set. At this point, the sequence `P` cannot be further extended, and so it can be printed since it represents a complete permutation.

So we have described all ingredients of the algorithm except for how we will represent `P` and `R`. Clearly we can use arrays to store these. But note that if we our original problem is to print permutations of $\{$`0,...,n-1`$\}$, then the sum of the lengths of `P` and `R` is always `n`. Hence we can store `P,R` together in a single array of length `n`. Our convention will be to store `P` first and `R` after that in the array. So if we state the length `pL` of `P` we will know which part of the array is `P` and which is `R`.

The recursive procedure gPerm is as follows. It uses a `swap` function, which merely exchanges the values of its arguments. This is in fact the `swap2` function from Section 11.2.

```
void gPerm(int A[], int n, int pL)
// A[0..pL-1] = P, A[pL..n-1] = R
// print all permutations of A,  P = A[0..pL-1] fixed
{
  if(pL==n){                    // base case: R is empty
    for(int i=0; i<n; i++)
      cout << A[i] << " ";
    cout << endl;
  }
  else{
    for(int i=pL; i<n; i++){   // For each element A[i] of R
      swap(A[i],A[pL]);        // extend P by appending A[i]
      gPerm(A,n,pL+1);         // recurse
      swap(A[i],A[pL]);        // undo the movement
    }
  }
}
```

The function works almost exactly as we discussed earlier. The base case, R being empty, corresponds to pL==n. In this case the array contains the permutation, which we print. For the recursion we must call with P extended by 1 character. So the character which extends P must be moved to position pL, since at the beginning of the call P extends from position 0 to position pL-1 of A. After the recursive call, we must undo the movement to the position pL.

Our main program is

```
main(){
  int n; cin << n;
  int A[n];          // the array in which P,R are to be stored.
  int pL=0;          // initially P is empty, so its length pL = 0.
  for(int i=0; i<n; i++) a[i] = i;
                     // The array only stores R, which is 0..n-1 initially
  gPerm(A, n, pL);  // array name, length of array, length of P
}
```

13.8 Exercises

1. Write a program that reads in an integer from the keyboard and prints it out in words. For example, on reading in 368, the program should print "three hundred and sixty eight".

2. For this exercise it is important to know that the codes for the digits are consecutive, starting at 0. Further '8' - '0' is valid expression and evaluates to the difference in the code used to represent the characters, and is thus 8. To clarify, if we execute

   ```
   char text[10] = "1729";
   int d = text[1] - '0';
   ```

   Then d will have the value 7. Use this to write a function that takes a char array containing a number and return an integer of the corresponding magnitude.

3. Extend the marks display program of Section 12.2.2 to use names rather than roll numbers. At the beginning, the teacher enters the number of students. Then the program must prompt the teacher to enter the name of the student, followed by the marks. After all names and marks have been entered, the program then gets ready to answer student queries. Students enter their name and the program prints out the marks they have obtained.

4. Write a function which takes a sequence of parentheses, open and closed, of all types, and says whether it is a valid parenthesization. Specifically, parentheses should be in matching pairs, with the opening parenthesis before the closing, and if a pair of parentheses contains one parenthesis from another pair, then it must also contain the other parenthesis from that pair.

5. Write a "calculator" program that takes 3 command line arguments in addition to the name of the executable: the first and third being `double` values and the second being a single `char`. The second argument must be specified as an arithmetic operator, i.e. `+, -, *` or `/`. The program must perform the required operation on the two numbers and print the result.

6. Write the function `solveLSE` of Section 13.5.

7. Write a function which given a square matrix returns its determinant. You should NOT use the recursive definition of determinant, but instead use the following properties:

   - Adding a multiple of one row to another leaves the determinant unchanged.
   - Exchanging a pair of rows causes the determinant to be multiplied by -1.
   - The deteminant of an upper triangular matrix (all zeros below the diagonal) is simply the products of the elements on the diagonal.

   If the first element of the first row is a zero, then exchange rows so that it becomes non zero. Then add suitable multiples of the first row to the other rows so that the first column is all zeros except the first row. Similarly produce zeros below the diagonal in the second column, and so on.

8. Write the program to find shortest paths as discussed in Section 13.6.

9. Let p be a permutation of integers from 0 to $n-1$ for some n. Define $V(p)$ to be the integer obtained by concatenating the sequence p. We will say that a permutation p is *lexicographically* smaller than another permutation q if $V(p) < V(q)$. The permutation generation algorithm given in Section 13.7 does not generate the permutations in lexicographically increasing order. Modify it to do so.

10. Write a program that takes a permutation p of integers from 0 to $n-1$ and returns the lexicographically next permutation. Hint: try out a few permutations to deduce the relationship between a permutation and the lexicographically next permutation.

11. In the 8 queens problem, you are required to place 8 queens on a chessboard so that no queen captures another. For those who do not know chess: a chessboard has 8 rows and 8 columns, and two queens capture each other if they are in the same row, or in the same column, or in the same (not necessarily principal) diagonal. Write a program to solve the analogous n-queens problem. Hint: modify the permutation generation program. Your program should stop after printing k solutions (if any), where k is specified by the user in addition to n.

12. You might have observed that most physical objects are designed to have smoothed rather than sharp corners. One way to smooth a corner is to locally inscribe a circular arc that is tangential to edges forming the corner. However, other curves are also often used. One such family of curves are the *Bezier curves*, which have been used in the design of automobile bodies, for example. A Bezier curve of order n is a parametric curve defined by using n control points $p_1, \ldots, p_n$. The parameter which we will denote

by t varies between 0 and 1, and for each value of t we get one point $B_{p_1,\ldots,p_n}(t)$. This point can be determined recursively as follows. First, the base case:

$$B_p(t) = p$$

To compute $B_{p_1,\ldots,p_n}(t)$, we first determine points $q_1, \ldots, qn - 1$, where

$$q_i = tq_i + (1-t)q_{i+1}$$

i.e. q_i is the point dividing the line segment $p_i p_{i+1}$ in the ratio $t : 1 - t$. Now we have:

$$B_{p_1,\ldots,p_n}(t) = B_{q_1,\ldots,q_{n-1}}(t)$$

Write a program which receives points on the graphics canvas and plots a Bezier curve through them. Experiment for different values of n.

Chapter 14

Structures and Classes

By now, you have learnt enough programming language features to be able to write any program you might wish to write. However, being able to write any program is different from being able to write any program *conveniently.*

Suppose for example that you are writing a program involving sets. Presumably, your program will need to keep track of several sets, and perhaps perform operations on them, say taking the union of two sets. Wouldnt it be nice if you could refer to sets in your program by giving them names and write something like `c = union(a,b);`, where `a,b` are sets, and where we want `c` to become the set which is the union of `a,b`? Basically, just as we can declare variables `p,q,r` of type `double`, perhaps we should be able to declare variables `a,b,c` of type `set`. Just as we routinely perform arithmetic on `double` data, we should be able to perform set operations on `set` type data.

This is not to say that C++ should supply us with a built-in data-type for representing sets, or a built-in function for computing union. The function will we written by us, but the language should merely supply us the facilities to write such functions and define such data types. More generally, it might be desirable to have data types for representing any class of entities that is important in the program that we are writing. For example, if we are writing programs about books in a library, it might be useful to define a `Book` data type; variables of type `Book` could then be conveniently used to hold information about a book. And of course, we should be able to operate on such objects in a convenient manner, say by using functions which we would write. This idea is the beginning of an approach to programming called *Object-oriented programming.* This chapter takes a step towards understanding this approach.

The first step in this approach is to collect together all the information concerning an entity and be able to refer to the collection by a name. This is somewhat like an array; an array name does refer collectively to lots of elements; except that now we want a name to refer to a collection of elements which might be of different types. For example, for a book we might want the collection to contain the name, name of the author, price, a library number, information about who has borrowed it and so on. A *structure*, as we will discuss in this chapter provides what we want: it allows us to group together data of different kinds into a single collection which we can collectively refer to by a single name. Using structures will turn out to be very natural for many applications.

We begin by discussing the basic ideas of structures. We will show several examples, and then discuss at length a structure using which 3 dimensional vectors can be nicely represented and elegantly manipulated in programs. We also discuss how to build a structure to represent the queue like functionality we saw in Section 12.2.6. We then discuss some advanced features of structures, which finally takes us to the notion of classes.

14.1 Basics of structures

The word *structure* is used to denote a variable (i.e. contiguous region of memory) containing inside it a collection of variables of fundamental data types. The variables in the collection are said to be members of the structure.

Before we can define structure variables, we must first define a *structure-type*.

```
struct structure-type {
  member1-name member1-type;
  member2-name member2-type;
  ...
}
```

This statement says that the name `structure-type` will denote a new type of variable, or a new *data type*. A variable of type `structure-type` will be a collection of sub-variables or *members* whose names and types are as given. The rules for choosing names for structure types or members are the same as those for ordinary numerical variables, but it is often customary to capitalize the first letters of structure names, which is a convention we will follow.

As an example, here is how we might define a structure to store information about a book.

```
struct Book{
  char title[50];
  char author[50];
  double price;
  int accessionNo;
  bool borrowed;
  int borrowerNo;
};
```

Note that a structure definition does not by itself create variables or reserve space. But we can use it to define variables as follows.

```
Book pqr, xyz;
```

This statement is very similar to a statement such as `int pqr, xyz;` – it causes variables `pqr` and `xyz` to be created, of type `Book`. Space is also reserved in memory for each variable. In this space the various members of the structure are stored. Assuming 4 bytes are used to store an `int` and 8 for a `double`, we will need 16 bytes to store the members `accessionNo`, `borrowerNo`, and `price`, and 50+50 bytes to store the members `title` and `author`. A `bool` data type will typically be given 1 byte. So a total of 117 bytes has to be reserved each for pqr and xyz. The number of bytes that effectively get used might be larger, because there may be restrictions, e.g. on many computers it is necessary that the starting address of a variable must be a multiple of 4.

A member of a structure variable can be referred to by joining the variable and the member name with a period, e.g. xyz.accessionNo. Such references behave like variables of the same type as the member, and so we may write:

```
xyz.accessionNo = 1234;
cin.getline(pqr.title,50);
```

The first statement will store 1234 in the 4 bytes of the 117 bytes reserved for xyz. In the second statement, the reference pqr.title refers to the first of the two char arrays in pqr. Just as we can read a character string into a directly defined char array, so can we into this member of pqr.

We can initialize structures in a manner similar to arrays. Assuming Book defined as above we might write

```
Book b = {"On Education", "Bertrand Russell", 350.00, 1235, true, 5798};
```

This will copy elements of the initializer list to the members in the structure.

Here is a structure for representing a point in two dimensions.

```
struct Point{
  double x;
  double y;
};
```

We may create instances of this structure in the manner described before, i.e. by writing something like Point p1;. We are allowed to have one structure be contained in another. Here for example is a structure for storing information about a circle,

```
struct Circle{
  Point center;
  double radius;
}
```

Now the following natural program fragment is quite legal.

```
Circle c1;
c1.center.x = 0.5;
c1.center.y = 0.9;
c1.radius   = 3.2;
```

We could also have accomplished this by writing:

```
Circle c1 = {{0.5,0.9},3.2};
```

One structure can be copied to another using an assignment. For example, assuming c1 is as defined above, we can further write:

```
Circle c2;
c2 = c1;
```

This causes every field of c1 to be copied to the corresponding field of c2. In a similar manner we could also write:

```
Point p = c1.center;
c2.center = p;
```

The first statement copies every member of c1.center to the corresponding members of p. The second copies every member of p to the corresponding member of c2.center.

We finally note that variables can be defined in the same statement as the definition of the structure. For example, we could have written

```
struct Circle{
  Point center;
  double radius;
} C1;
```

which would define the struct Circle as well as an instance C1.

14.1.1 Visibility of structure types and structure variables

If a structure type is going to be used in more than one function, it must be defined outside both the functions. The definition must textually appear before the functions in the file.

The rules for accessing structure variables are the same as the rules for variables of the fundamental data types (Section3.8). Thus in the block in which a variable is defined, it can be used anywhere following its definition. Also, it might shadow names defined in blocks outside the block in which the definition appears, or it might in turn be shadowed by names defined in blocks contained in the current block.

14.1.2 Structures and functions

It is possible to pass structure variables to functions. The key point to be noted is that the name of a structure denotes the content of the associated memory, unlike the name of an array, which denotes the address of the associated memory. Just like numerical data types, structures can be passed by value (Section 9.1.1), or by reference (Section 11.2). We see examples next. It is also possible to return a structure from a function. We will see examples of this in Section 14.2.

In Section 12.2.7, we wrote a program to decide if any pair of circles from a given set of circles intersect. A key part of the program was the code to determine whether two circles intersect. Assuming the definition of structure Circle as given above, we can check for intersection using the following function.

```
bool intersects(Circle ca, Circle cb){
  if(pow(ca.center.x-cb.center.x,2) + pow(ca.center.y-cb.center.y,2)
     <= pow(ca.r+cb.r,2)){
```

```
    return true;
  }
  else return false;
}
```

The function could be called from the following fragment of the main program:

```
Circle c1 = {{0,0},3.2}, c2={{0,5},2.0};
bool q=intersects(c1,c2);
```

and it would set q to true. Note that for the procedure call, the entire memory region of c1,c2 is copied into the region of ca,cb in the activation frame of the call. Note further that when the function execution ends, only the result, in this case just a single bit, is copied back. Thus even if the function were to change ca.center.x, the circle c1 of the calling program would not be affected. If we want the values of the arguments to change, we would have to mark the corresponding parameter as a reference parameter. Here is an example.

```
void expand(Circle & c, double factor){
  c.r = c.r * factor;
}
```

This would cause the member c.r to scale up by a factor factor, i.e. the circle center would not change but the circle would just become bigger. So if this is called from the main program as expand(c1,2.0), with c1 as defined above, then the radius of c1 would become 6.4.

Besides enabling the argument to be changed, there is another important benefit of passing an argument by reference. The value of the argument is *not* copied over from the calling function to the called function, instead only the reference (a single word, typically) is sent. Thus this will likely be faster, if the structure is very large.

14.1.3 Pointers to structures

The "address of" operator & and the dereferencing operator * can be used with structures and pointers to structures respectively. Thus we could write the expand function using pointers as follows.

```
void expand2(Circle *c, double factor){
  (*c).r = (*c).r * factor;
}
```

As you would find natural, the first parameter c, is declared to be of type Circle*, i.e. pointer to Circle. The expression *c dereferences the pointer getting back the circle itself whose address was passed. The member r of this circle is modified. This could be called from a main program such as follows.

```
main(){
  Circle c1={{0,0},3.2};
  expand(&c,2.0);
}
```

If structures are passed to functions by pointers, then expressions such as `(*pqr).` where `pqr` is a structure pointer appear very frequently, i.e. whenever a member of the structure pointed to by `pqr` is to be accessed. So for this, a different operator is defined in C++. Instead of writing `(*pqr).abc` to access member `abc` of `(*pqr)`, you may simply write `pqr->abc`. Thus the function `expand2` could also be written as follows.

```
void expand2(Circle *c, double factor){
  c->r = c->r * factor;
}
```

If for some reason you wanted to refer to the x coordinate of the center of the circle to which you have a pointer `c` as above, you would write it as `c->center.x` as you might expect.

Clearly, the `->` operator is more readable than the derereferencing and "`.`" combination. However, it is even better not to use pointers if possible and pass the structure by reference instead.

14.1.4 Constant references

We have mentioned that passing structures by reference has the benefit of not having to copy the structure at the time of the call. So indeed we should consider passing structure arguments by reference to every function. When our goal is to merely avoid copying and we do not want to change the reference parameter, it is useful to indicate our intent. In C++, this can be done by using the prefix `const` before the parameter declaration. So for example we might write

```
bool intersects(const Circle &ca, const Circle &cb){
  if(pow(ca.center.x-cb.center.x,2)+ pow(ca.center.y-cb.center.y,2)
     <= pow(ca.r+cb.r,2))
  return true;
  else return false;
}
```

This style has the benefit of preventing the arguments from being copied, and in addition conveniently tells a human reader that the parameters will not be modified. It also has another desirable effect which will be discussed in Section 14.2.2

14.1.5 Arrays of structures

Note that we can make arrays of any data type. For example, we could make an array of circles or books if we wished.

```
Circle c[10];
Book library[100];
```

We can refer to members of the elements of the arrays in the natural manner. For example, `c[5].center.x` refers to the x coordinate of the center of the fifth circle in `c`. Similarly `library[96].title[0]` would refer to the starting letter in the `title` of the 96th book in `library`.

Let us now write the program from Section 12.2.7 using an array of circles, rather than separate arrays for storing the x,y coordinates of the center and for the radius. We assume that the function `intersects` has been defined as above, and before that the structures `Point` and `Circle` have been defined. Following the function, the main program can be written as follows.

```
main(){
  int n; cin >> n;
  Circle circles[n];

  for(int i=0;i<n;i++)          // read in all data.
    cin >> circles[i].center.x >> circles[i].center.y >> circles[i].r;
                                // Find intersections if any.
  for(int i=0; i<n; i++){
    for(int j=i+1; j<n; j++){
      if (intersect(circles[i], circles[j]))
        cout << "Circles " << i << " and " << j << " intersect." <<endl;
    }
  }
```

14.2 Representing 3 dimensional vectors

In the next chapter we will see a program which deals with motion in 3 dimensional space. This program will deal considerably with 3 dimensional vector quantities such as positions, velocities, and accelerations. So we will design a structure which makes it convenient to represent such quantities.

A vector in 3 dimensions can be represented in many ways. For example, we could consider it in Cartesian coordinates, or in so called spherical coordinates, or cylindrical coordinates. For simplicity, we consider the first alternative: Cartesian coordinates. Thus we will have a component for each spatial dimension. Clearly our structure must hold these 3 coordinates. We will call our structure `V3` and it can be defined as:

```
struct V3{
  double x,y,z;
};
```

We should put this outside the main program. If the program is organized into many files, this should go into a header file, which can then be included in all other files which need this structure.

First we will write a function which will construct a `V3` structure given values of the 3 components.

```
V3 make_V3(double p, double q, double r){
  V3 v;
  v.x = p;
  v.y = q;
  v.z = r;
  return v;
}
```

Note that this returns v, which will be *copied* back to the calling program. The real use of this function is in the following function which allows us to add two V3 numbers.

```
V3 sum(V3 a, V3 b){
  return make_V3(a.x+b.x, a.y+b.b.y, a.z+b.z);
}
```

Thus we can now write a code fragment like the following.

```
V3 a={1,0,5}, b={3,4,2}, c;
c = sum(a,b);
```

This will make c be the vector with components 4,4,7. Similarly we can write functions which will perform other arithmetic operations. A natural function that will be needed is the length of a vector.

```
double length(V3 a){
  return sqrt(a.x*a.x + a.y*a.y + a.z*a.z);
}
```

Ideally, you might wonder, wouldnt it be nicer if we could just write expressions such as c=a+b; rather than c=sum(a,b);? This is easily arranged! We can *overload* operators so that they will work with our vectors!

14.2.1 Operator overloading

The addition operator, +, is not automatically defined for all data types. But we can define it! For this note first that C++ really treats operators as functions of two arguments. For ordinary functions, the arguments are supplied in parentheses, separated by commas. However, for an operator such as +, the operator appears *in* between the arguments, and hence it is called an *in*fix operator. To define the + operator for V3 numbers, you only need to note that the name associated with + is operator+, which is what you need to overload. Thus it suffices to write the following.

```
V3 operator+ (V3 a, V3 b){
  return make_V3(a.x+b.x, a.y+b.y, a.z+b.z);
}
```

Similar definitions can be made for - by writing operator- instead of operator+. This is left as an exercise. It is common to multiply a vector by a scalar. This merely scales each component. If we want to be able to write this using the operator * we could arrange it by writing the following.

```
V3 operator*(V3 a, double t){
  return make_V3(a.x*t, a.y*t, a.z*t);
```

Notice that if s,u,a are vectors denoting the distance covered, initial velocity and (uniform) acceleration, and t denotes the time, then we can compute s by writing s = u*t + a*t*t*0.5;. This succinctly expresses the kinematics formula $s = ut + \frac{1}{2}t^2$.

An interesting case is that of the << operator. This is also a binary operator, but in this case the first operand is of type ostream. We have seean earlier that stream variables cannot be copied, so they must be passed by reference. Further, if we want to write expressions such as cout << z1 << z2;, which really should be read as (cout << z1) << z2;, the first part, cout<<z1 should evaluate to cout. This is what the following definition implements.

```
ostream & operator<<(ostream & ost, V3 v){
  ost << "(" << v.x << ", "<< v.y << ", "<< v.z << ")";
  return ost;
}
```

Note that the value returned would normally be copied back. Again, because copying data of type ostream is not allowed, the value is returned by reference, hence the type of the return value is ostream &.

Now if a,b were defined as in the main program above, then you could write cout << "a: "<< a <<" b: "<< b << endl; and this would print

```
a: (1, 0, 5) b: (3, 4, 2)
```

The exercises ask you to similarly overload the >> operator so that a triple of numbers can be read into a V3 variable.

14.2.2 Pass by value or by reference

In the entire discussion above, we have passed the V3 parameters by value. We should consider passing by reference, because that would prevent the need to copy the argument values to the parameters. Further, we should add the qualifier const wherever we know that the corresponding parameter will not be modified. As an example, we could have written:

```
V3 operator+ (V3 const& a, V3 const& b){
  return make_V3(a.x+b.x, a.y+b.y, a.z+b.z);
}
```

We noted in Section 11.2.1 that if an argument is passed by reference, then it cannot be constant. However, that remark does not apply when the reference is declared const. Since the parameter is constant, there is no question of it getting modified, and hence the problems as mentioned in Section 11.2.1 would not arise. In other words, with the above definition, we can write the call make_V3(1,0,5) + make_V3(3,4,2), and it would return the V3 vector (4,4,7) as expected.

14.2.3 Putting it together

We can use the functions and structure that we have developed to write a main program as follows.

```
main(){
  V3 u,a;
  double t;
  cin >> u.x >> u.y >> u.z >> a.x >> a.y >> a.z >> t;
  cout << u*t + a*t*t*0.5 << endl;
}
```

Following the discussion in Section 14.1.1 we place the definition of the structure V3 at the top of the file. Then we place all the functions, and then finally the main program. We can then compile this file and run it. This will ask the user to give the initial velocity, the acceleration, and the time duration, and the distanc covered will be printed.

14.3 Structures: advanced features

We could think of a structure as merely a mechanism for managing data; we organize data into a collection rather than have lots of variables lying around. However, once you define a structure, it becomes natural to write functions which manipulate the data contained in the structures. You might say that once we defined V3, it is almost inevitable that we write functions to perform vector arithmetic and compute the Euclidean length. Had we defined a structure to represent some other entity, say a book (in a library), we might have found it useful to write a function that performs the bookkeeping needed when a book is borrowed.

Indeed, you might consider these functions to be as important to the structure as are members of the structure. So perhaps, should we make the functions a part of the structure itself?

The definition of structures you have seen so far really comes from the C language. In the more modern definition of structures, as it is in the C++ language, the definition of structures has been extended so that it can also include functions. At a high level, the more general definition of a structure is the same as before.

```
struct structure-type {
  member-description1
  member-description2
  ...
}
```

But, now, a `member-description` may denote a `member-function`, in addition to being able to denote a pair `member-type member-name` as before. Member functions can be special and ordinary. There can be two kinds of special member functions, the so called `constructor` and `destructor` functions. In addition there can be as many ordinary member functions as you want. We will shortly describe in general how member-functions are specified and called, but let us first see an example. Here is how our structure V3 will appear in this new definition style.

```
struct V3{
  double x,y,z;
  V3(double p, double q, double r){     // constructor 1
```

```
    x = p;
    y = q;
    z = r;
  }
  V3(){                              // constructor 2
    x = 0;
    y = 0;
    z = 0;
  }
  double length(){                   // ordinary member function length
      return sqrt(x*x + y*y + z*z);
  }
  V3 operator+ (V3 b){               // ordinary member function operator+
    return V3(x + b.x, y+b.y, z+b.z);
  }
  V3 operator- (V3 b){               // ordinary member function operator-
    return V3(x-b.x, y-b.y, z-b.z);
  }
  V3 operator* (double t){           // ordinary member function operator*
    return V3(x*t, y*t, z*t);
  }
};
```

This contains two constructor functions, and the ordinary member functions `length`, `operator+`, `operator-`, `operator*`. There is no destructor function; we will discuss destructors later when we need to.

14.3.1 Constructor functions

A constructor function is used to create an instance of the structure, analogous to the `make_V3` function we had earlier. In general, the `member-description` for a constructor function for a structure with name `structure-name` has the following form.

```
  structure-name (parameter1-type parameter1, parameter2-type parameter2,
                  ...){ body };
```

In this, `structure-name` is the return-type as well as the name of the function. You can specify as many constructors as you want, so long as the list of parameter types are different for each constructor. Constructors are used for defining variables of the given structure type. They may be called in the natural manner, by writing:

```
structure-name(argument1, argument2, ...)
```

Here is an example in which we create two variables by respectively calling our two constructors.

```
V3 vec1=V3(1.0,2.0,3.0), vec2=V3();
```

As you will see this will create a variable `vec1` with its `x,y,z` components set to 1.0,2.0,3.0, and a variable `vec2` with all its components set to 0.

A call to a constructor executes as usual by creating an activation frame. Then the arguments are evaluated and are copied to the corresponding formal parameters. Then something unusual happens: a nameless variable of type `structure-name` is created. Its data members can be accessed in the body of the constructor by using the member names themselves, i.e. without using the "." notation. Next, the body of the constructor is executed and when execution finishes, this nameless variable is returned as the result.

Let us now see how the call `V3(1.0,2.0,3.0)` in our example would execute. Clearly, this is a call to constructor 1, since there are 3 arguments. An activation frame is created and the argument values, 1.0, 2.0, 3.0 are copied to the corresponding formal parameters `p,q,r`. As noted a nameless structure `V3` is also constructed. Then the body of constructor 1 starts execution. The first statement of the body, `x = p;` sets the `x` member of the nameless structure to the value of the parameter `p`. Similarly, the members `y` and `z` are set to the values `q` and `r` respectively. Following this, the nameless object is returned as the result. This returned value, in our example, is copied to the variable `vec1`. Clearly, `vec1` will have 1.0, 2.0 and 3.0 as its `x,y,z` members. The second constructor executes similarly. It takes no arguments, but on examining its body, you will see that it sets all 3 members `x,y,z` all to 0. Thus the variable `vec2` will have all its 3 members be 0.

The more customary way of writing the above statement is

```
V3 vec1(1.0,2.0,3.0), vec2;
```

As you can see, what follows the variable name is used as the argument list to call an appropriate constructor. If the argument list is empty you are expected to not give it at all, as in the construction of `vec2` above. Note that if you write `V3 vec2();` it means something quite different: it declares `vec2` to be a function that takes no arguments and returns a result of type `V3`, as we discussed in Section 9.7.1.

If one structure is nested inside another, then the constructor executes slightly differently. This and other nuances are discussed in Section 14.4.

14.3.2 Ordinary member functions

The ordinary member functions `length`, `operator+` and so on in `V3` will play the role of similarly named functions of Section 14.2. In general, the `member-description` of an ordinary member function for a structure with name `structure-name` has the following form.

```
return-type function-name (parameter1-type parameter1, parameter2-type
   parameter2, ...){body}
```

A member function is expected to be called *on* a variable of the given structure type, using the same "." notation used for accessing data members. Suppose `var` is a variable or expression of type `structure-name`. Then the member function `function-name` is called on it as:

```
var.function-name(argument1, argument2, ...)
```

You are indeed expected to think of a member function call as being similar to accessing a data member. For the execution, the variable `var` treated just like an argument. The execution of the call happens as follows.

1. The expressions `var, argument1, argument2, ...` are evaluated.

2. An activation frame is created for the execution.

3. The values of the arguments `argument1,...` are copied over to the corresponding parameters.

4. Each member function is deemed to have an implicit *reference* parameter of type `structure-name`. The variable `var` is considered as a reference argument for this implicit parameter.

5. The body of the function is executed. You can think of the implicit parameter as providing a context, i.e. inside the body, the names of the data members by themselves are considered to refer to the corresponding members of the implicit parameter. Note that since the implicit parameter is a reference parameter, any changes we might make get reflected in `var`.

Let us now see an example.

```
V3 p(1,2,3), q(3,4,5), r,s;
cout << p.length() << endl;
```

In the call `p.length()`, the implicit parameter will be `p`, and hence the names `x,y,z` in the body of `length` will refer to `p.x,p.y,p.z`, which have been respectively initialized to 1,2,3 in the first statement. Thus, the statement `return sqrt(x*x + y*y + z*z);` will return `sqrt(1*1+2*2+3*3)`, i.e. $\sqrt{14}$.

14.3.3 Overloading operators

Suppose we have an expression involving an operator, say

```
lhs + rhs
```

where `lhs` is a `struct`. Then we can define how this expression is to be executed by defining a member function `operator+` which takes a single argument. If such a function has been defined, the compiler views the expression `lhs + rhs` as the member function call

```
lhs.operator+(rhs)
```

which is executed, and its value is considered to be the value of the expression `lhs + rhs`. For some operators `+`, we can define `operator+` as an ordinary function, as discussed in Section 14.2.1. Most operators can be overloaded as member functions or ordinary functions. However, some operators, viz. `=` (assignment), `[]` (array subscript), `->` (member access), and `()` (function call), can only be implemented as member functions. We will see examples of some of these in Section ??, Section ?? and Section 18.2.4.

Our definition of `V3` already contains the member function `operator+`. So suppose now that you wrote the following code.

```
r = p.operator+(q);
s = p+q;          // more customary form
```

These statements will execute as follows. In the call `p.operator+(q)`, `p` will be the implicit parameter, and the the parameter `b` in the body of the member function definition will take the value `q`. Thus the names `x,y,z` will refer to `p.x,p.y,p.z`, i.e. 1,2,3. Also `b.x,b.y,b.z` will refer to `q.x,q.y,q.z`, i.e. 3,4,5. Thus the statement statement `return V3(x + b.x, y + b.y, z + b.z)` will cause `V3(4,6,8)` to be returned. Thus `r` will become the structure with components 4,6,8. The more customary way of using the function `operator+` is of course to write the binary infix operator `+`, as is shown in the next line, `s=p+q;`. This will cause `s` to also be set to a vector with components 4, 6, 8.

14.4 Additional issues

We now discuss some aspects of structure definition not directly seen in the `V3` example codes discussed so far.

14.4.1 Call by reference and `const` declaration

It is possible to pass arguments to member functions by reference. The mechanism for this is exactly the same as for ordinary functions (Section 11.2). Further note that we can declare parameters to be `const`, i.e. that they will not change during execution of the member function. Note that a member function is invoked on some instance of the `struct`, it is possible that the member function does not change that struct either. For this, we need to place an additional `const` keyword, after the parameter list of the function but before the body.

Thus we could have defined `operator+` of `V3` as follows.

```
V3 operator+ (V3 const & b)  const {  // notice there are 2 const's
  return V3(x+b.x, y+b.y, z+b.z);
```

The `const` in the declaration of the parameter `b` merely states that `b` will not change. The second `const`, after the parameter list says that the implicit argument does not change either.

Similar changes should be made to the other member functions in `V3`.

14.4.2 Default values to parameters

Parameters to member functions can also be given default values. For example, we could have bundled our two constructors for `V3` into a single constructor by writing

```
V3(double p=0, double q=0, double r=0){
  x = p;
  y = q;
  z = r;
}
```

Now you could call the constructor with either no argument, or upto 3 arguments – parameters corresponding to arguments that have not been given will get the default values, in this case 0. Note that if you include our new constructor in the definition, you cannot include any of the constructors we gave earlier. Say you specified the new bundled constructor and also constructor 2. Then a call `V3()` would be ambiguous, it would not be clear whether to execute the body of constructor 2, or the body of the new constructor in which all 3 parameters are initialized to their specified defaults.

14.4.3 Default Constructor

You may wonder what happens if our structure definition contains no constructor at all. In this case, C++ supplies a *default constructor*. This constructor takes no arguments, and its body is empty: for `V3` the default constructor would have been exactly like our constructor 2, but with an empty body. Such constructors would be supplied by C++ for all the `struct`s we defined in Section 14.1, since we gave no constructors for them.

The term *default constructor* is actually used more generally: it has come to mean a constructor that can be called with no arguments, even if such a constructor has been explicitly defined by the programmer. Thus for `V3` our constructor 2, as well as our bundled constructor would be called default constructors. Of course, as we remarked earlier, 2 default constructors cannot be specified simultaneously for any structure.

Earlier we remarked that “Constructors are used for defining variables” – this should be read in a strong sense: variables can *only* be created using constructors. Thus it is only because of the default constructor supplied by C++ that we were able to construct structure variables in Section 14.1, for example. We note further that a default constructor is needed if you wish to define arrays of a structure, because each element of the array will be constructed only using the default constructor.

Note that C++ does not supply a default constructor if you give any constructor whatsoever. So if you define a non-default constructor (i.e. a constructor which must take at least one argument), then the structure would not have a default constructor. Thus you would not be able to create arrays of that structure.

The default constructor is important also when we nest a structure inside another. We discuss this next.

14.4.4 Constructors of nested structures

Consider the `Point` and `Circle` classes of Section 14.1, defined as

```
struct Point{double x,y;};
struct Circle{Point center; double radius;};
```

Consider what happens when we execute

```
Circle c;
```

As discussed above, the default constructor for `Circle` would be called. Since we did not supply a constructor, C++ will create one for us. Note however, that this constructor must construct all the members of `Circle`. To accomplish this, the constructor created by C++

will call default constructors of all the members as well. So in our case, the C++ constructed constructor for `Circle` will call the default constructor for `Point`.

Suppose now we change our definition of Point so that it has a constructor.

```
struct Point{
  double x,y;
  Point(double p, double q){x=p; y=q;}
};
```

Note that this constructor takes two arguments, and hence is not a default constructor. Further, because a constructor is given for `Point` C++ will not create any constructors for `Point`. The first observation, then, is that now you would not be write `Circle c;`. This is because the constructor created for `Circle` would expect a default constructor to be present for `Point`.

To overcome this problem, you might decide to write your own constructor for `Circle`, and not rely on C++ to supply one. Here is a possible attempt.

```
struct Circle{
  Point center;
  double radius;
  Circle(double x, double y, double r){
    center = Point(x,y);
    radius = r;
  }
};
```

Unfortunately, this attempt does not work. The reason for this is a bit involved but worth understanding.

We said that *before* the constructor body executes, a nameless structure of type `Circle` is created, which is used as a context to the execution of the body of the constructor. The phrase "a structure of type `Circle` is created" is interpreted very strongly – if the structure is created, its members must also be in a "created state", i.e. the constructors must already have been called for them. Thus a constructor will already have been called to construct every member of `Circle` even before the `Circle` constructor body starts execution!

In the absence of additional information, the members will be created using their default constructors. For the member `radius`, you can assume that C++ will have created a default constructor which does nothing, so there is no problem for this member.[1] But for the member `center`, C++ will attempt to look for a default constructor, not find it, and generate an error.

The problem can be solved using *initialization lists.*

14.4.5 Initialization lists

The correct code which will use the non-default constructor of `Point` is as follows.

[1] Or you can equally well assume that members which are fundamental data types do not need to be constructed.

```
struct Circle{
  Point center;
  double radius;
  Circle(double x, double y, double r) : center(Point(x,y)), radius(r)
  {
  // empty body
  }
};
```

The text following the : to the end of the line in the above code is an *initilization list.* The initialization list of a constructor says how the data members in the nameless structure should be constructed before the execution of the constructor itself can begin.

Thus in this case the code says that `center` should be constructed using the constructor call `Point(x,y)`, where `x,y` are from the parameter list of the `Circle` constructor. Similarly the member `r` of the `Circle` being constructed is assigned the value `r`. In general, the initialization list consists of comma separated items of the form

`member-name(initializing-value)`

This will cause the member `member-name` to be initialized directly using `initializing-value`. If the initializing value calls a constructor, then instead of writing out the call, just the comma separated arguments could be given. Thus, for our `Circle` constructor, the initialization list could also have been:

`center(x,y), radius(r)`

Note that in our example, all the work got done in using the initialization lists, so the body is empty. Note that we could choose to initialize only some of the members using the initialization list and initialize the others in the body, if we wish.[2]

14.4.6 Constant members

Sometimes we wish to create structures in which the members are set at the time of the initialization, but not changed subsequently. This *write-once* strategy of programming is very comforting: it is easier to reason about a program if you know that the values once given do not change.

If we want our `Point` structure to have this property, then we would write it as follows.

```
struct Point{
  const double x,y;
  Point(double x1, double y1) : x(x1), y(y1)
  { // empty body }
}
```

Notice that we have given values to members `x,y` using initialization lists. This is treated as initialization of the members, and not as assignment to the members. On the other hand, in the constructor of the previous section, the values were assigned in an assignment inside the body – that is not allowed if the member is declared `const`. So initialization lists are useful for this as well.

[2] Whenever possible you should use initialization through initialization lists, because it is likely faster.

14.4.7 Static data members

Suppose you wish to keep a count of how many `Point` objects you created in your program. Algorithmically, this is not difficult at all; we merely keep an integer somewhere that is initialized to 0, and then increment it when we create an object. The question is: how should this code be organized.

First, we need to decide where to place the counter. It would seem natural that the counter be somehow associated with the `Point` type. This is indeed possible through the use of *static data members*, as follows.

```
struct Point{
  double x,y;
  static int counter;            // only declares
  Point(){
    counter++;
  }
  Point(double x1, double y1) : x(x1), y(y1){
    counter++;
  }
};
int Point::counter = 0;         // actually defines

int main(){
  Point a,b, c(1,2);
  cout << Point::counter << endl;
}
```

A static data member is a variable associated with a `struct` type. It is declared by prefixing the keyword `static` to the declaration. Note that while there will be a member `x` and a member `y` in every `Point` structure created through either of the constructors, there is only one copy of the variable `counter`. Inside the definition of `Point`, the variable `counter` can be referred to by using the name `counter`, outside the definition a static variable must be referred to by prefixing its name by the `struct` name and `::`. So in this example we have used `Point::counter`.

There is a subtlety associated with static data members. The definition of the structure `Point` does not actually create the static data variables; a `struct` definition is merely expected to create a *type*, without allocating any storage. Hence we need the statement marked "`actually defines`" in the code above.

14.4.8 Static member functions

You can also have static member functions. For example, in the above code we could have added the following static function definition of `resetCounter` to the definition of `Point`.

```
  static void resetCounter(){ counter = 0; } // note keyword ``static''
```

Static member functions can be referred to by their name inside the structure definition, and by prefixing the structure name and `::` outside the definition. Further, static member functions are not invoked on any instance, but they are invoked by themselves. So we can write `Point::resetCounter()` in the main program if we wish to set `Point::counter` to 0.

Note that in non-static member functions we use the names of the non-static members by themselves to refer to non-static members of the implicit parameter, i.e. the object on which the non-static member function is invoked. However, for a static member function, there is no implicit parameter. Thus it is an error to refer to non-static members by themselves in the body of a static member function.

14.4.9 The `this` pointer

Inside the definition of any ordinary member function, the keyword `this` is a pointer to the implicit parameter, and to the nameless structure in case of constructors. Normally, we do not need to use this pointer, because we can get to the members of the nameless structure or the implicit parameter by using the names of the members directly. However, it should be noted that we could use `this` too, thus we could have written the `length` member function in `V3` as

```
double length(){
  return sqrt(this->x*this->x + this->y*this->y + this->z*this->z);
}
```

But of course this is not really a good use for `this`!

Suppose we wanted to have a member function `bigger` in `Circle` which would take another `Circle` and return the bigger of the two circles. The function would need to just compare the radii, and then return the circle with the bigger radius. The following member function code could be added to the definition of `Circle` above.

```
Circle bigger(Circle c){
  return (radius > c.radius) ? *this : c;
}
```

We must return the implicit parameter if its radius is bigger than the radius of the argument circle. Thus we return `*this`.

14.5 A queue data structure

We revisit the taxi dispatch program of Section 12.2.6. At the heart of this program are the ideas of using the array `board` as a *queue*. We will put all the data related to the queue into a single structure, which will have member functions which allow elements to be inserted and removed. This will have two benefits. First, it will be much easier to use this structure in other programs if we wish. Second, the logic of managing the structure will get separated from the logic of using it, as a result the program will become easier to read.

Clearly, the structure which we will call `Queue`, must contain the array `board` and the variables `emptyBegin` and `emptyEnd` of Section 12.2.6. There must be methods to insert an

```
#include <simplecpp>
const int QUEUESIZE=101;

struct Queue{
  int front;
  int nWaiting;
  int board[QUEUESIZE];
  Queue(){
    front=0;
    nWaiting = 0;
  }
  bool insert(int value){
    if(nWaiting == n) return false;  // queue is full
    board[(front + nWaiting) % n] = value;
    nWaiting++;
    return true;
  }
  int remove(){
    if(nWaiting == 0) return -1; // queue is empty
    int item = board[front];
    front = (front + 1) % n;
    nWaiting--;
    return item;
  }
};

int main(){
  Queue q;
  while(true){
    char command; cin >> command;
    if(command == ’d’){
      int driver; cin >> driver;
      if(!q.insert(driver)) cout << "Cannot register.\n";
    }
    else if(command == ’c’){
      int driver = q.remove();
      if (driver == -1) cout << "No taxi available.\n";
      else cout << "Assigning: " << driver << endl;
    }
  }
}
```

Figure 14.1: Taxi dispatch using `Queue`

element into the queue, and to remove elements from the queue. Figure 14.1 shows the code. You can note clear parallels between this and the code in Section 12.2.6. The initialization of `front, nWaiting` has moved to the constructor. The insert method checks first if the queue is full, exactly as in Section 12.2.6, and if so returns false (failure). Otherwise it returns `true` (success) and the value is stored in the queue. The variable `nWaiting` is incremented, just as in Section 12.2.6. The key difference is that the code of queue management: checking for empty, incrementation, is moved to the method `insert`. Likewise for the method `remove`. The net result is that the main program becomes simpler and easier to understand. The methods in `Queue` are also easy to understand, because their logic is not mixed with taxi dispatching.

14.6 Access Control

When a structure such as `Book` (Section 14.1), `V3` (Sections 14.2,14.3) or `Queue` (Section 14.5) is designed, the designer usually has very clear ideas as to what are proper uses of the structure and what are the improper uses. For example, it is unlikely that a function using the `Queue` structure such as the function `main` of Figure 14.1 will contain a statement `q.front=7;`. It is expected that users of `Queue` will only insert and remove elements, and for this they will use the provided `insert` and `remove` methods.

The situation is very similar to packaged devices sold on the market. If you buy a radio, it is expected normally that you would only operate the controls provided to you on its front panel; you would not, for example, put wires inside it and try to connect it to some other device. In fact, manufacturers expressly warn against such use: often the guarantee about correct operation given by the manufacturers is considered null and void if you as much as open the backside of the device.

Likewise, when a professional programmer designs a structure for your use, he/she would like to make a guarantee to you regarding how it will work. However, the guarantee will be valid only if you use the structure as described by the designer. This is like the contract view of functions described in Section 9.3. It is only stronger: C++ allows the designer to *prevent* certain kinds of accesses to the structure.

14.6.1 Access specifiers

You may designate each member of a structure as either being `private`, `public`, or `protected`. To do this we divide the members in the class into groups, and before each group place `public:`, `private:` or `protected:` as we want the members in the group to be considered. You may use as many groups as you wish. For example we may define the structure queue as:

```
struct Queue{
private:
  int front;
  int nWaiting;
  int board[QUEUESIZE];
public:
  Queue(){...}
  bool insert(int value){...}
  int remove(){...}
};
```

In this, all the members following `private:` until the `public:` are said to be private. Private members can be accessed only inside the methods of the class, and are not accessible outside the class definition (but also see Section 14.6.2). Thus we would not be able to write a statement such as `q.emptyBegin=7;` in our main program – the compiler would flag it as an error.

The members following `public:` on the other hand, are considered to be accessible by all. In other words, they can be used inside the class definition if needed, but also outside of it. In other words, the above ensures that outside of the definition, we can construct an instance of `Queue` (use the constructor), insert elements, or remove elements, but not look at the data members.

We will explain `protected` members later.

A very common idea is to make all data members private (or protected, as you will see later), and a carefully chosen set of function members public.

14.6.2 Friends

If you make some members of a `struct` private, then they can only be accessed inside the struct definition. Sometimes this is too restrictive.

Suppose we want to enable a `Queue` instance to be printed, i.e. we would like to write

```
Queue q;
...
cout << q;
```

As we discussed earlier, we can do this by overloading the function `operator<<`. This function will take two arguments, the output stream and a Queue instance. We would like the output stream to be the first argument. Hence this function cannot become a member function for `Queue`, since then the queue instance would have to come first. Thus we are forced to write the overloading function outside of the definition of `Queue`. This poses a problem because `operator<<` would need access to the private members of `Queue`, which is not allowed.

C++ allows you to overcome this difficulty. You go ahead and define the `operator<<` function as you wish, accessing the private members also.

```
ostream & operator<< (ostream & ost, Queue q){
  if(q.nWaiting == 0) cout << "Queue is empty.\n";
  else{
    for(int i=q.front; i != q.front + q.nWaiting;
i = (i+1) % QUEUESIZE)
      ost << i << ": " << q.board[i] << endl;
      }
  return ost;
}
```

To enable the function `operator<<` to access the private members of `Queue`, you put a line in `Queue` along with the member descriptions as follows:

```
struct Queue{
   ...
   friend ostream & operator<< (ostream &ost, Queue q);
   ...
}
```

This will declare `operator<<` to be a friend, which means that it is allowed to access the private members of `Queue`. In general, the line will read `friend function-declaration`.

Notice that you could have acheived the same effect by declaring all members to be public. However, that would allow all functions access; by making a function a friend, you provide selective access.

Note that the same function can be a friend of several structures, and several functions be a friend of the same structure. In fact, you can have one structure `A` be a friend of another structure `B`. This way, the private members of structure `B` can be used inside the definition of structure `A`. To do this you merely insert the line `friend A;` inside the definition of structure `B`.

14.7 Classes

A *structure* as we have defined it, except for a minor difference, is more commonly known in C++ as a *class.*

The small difference between the two is as follows. In a structure, all members are considered public by default, i.e. a member that is not in any group that is preceded by a specifier is considered public. In a class, all members are considered private by default. To get the latter behaviour, you simply need to use the keyword `class` instead of `struct` in the definition.

```
class Queue{
...
};
```

It is customary to use the term *object* to denote instances of a class.

In addition to the features considered in this chapter, there are a number of other features in classes/structures, the most notable of them being *inheritance*, which we will consider in the following chapters.

14.8 Header and implementation files

Quite often, a class (or struct) will be developed independently of the program that uses it, possibly by a different programmer. Thus we need a protocol by which the code that defines the class can be accessed by code in other files. Following our discussion of functions, it is customary to organize each class `C` into two files: `C.h` and `C.cpp`.

First, some important terms. It is customary to say that the body of each member-function provides an *implementation* of the member-function. In fact, the bodies of all member functions together are said to constitute an implementation of the class itself. When the implementation is given as a part of the class definition, it is said to be given *in-line*. However, when classes are large and developed independently, it is more customary to put the definition of a class `C` without out the implementation, into the file `C.h`, the so called header file. The implementation is put into the file `C.cpp`, using some special syntax. If there are any friend functions, their declarations can also put in `C.h`, and implementations in `C.cpp`. We show this using an example.

Consider our class `V3` of Section 14.3. We will show the files `V3.h` and `V3.cpp` for it. What we show here is slightly different from Section 14.3. We will make `V3` be a class, and declare the data members `x,y,z` as private, as is customary. Often, when a vector is created, we do not expect users to fiddle with individual coordinates, however, users may want to know the values of (but not modify) the components. For this we have provided additional member functions, often called `accessor` functions because they access the members. The file `V3.h` for all this would be as follows.

```
class V3{
private:
  double x, y, z;
 public:
  V3(double p=0, double q=0, double r=0);
  V3 operator+(V3 w);
  V3 operator-(V3 w);
  V3 operator*(double t);
  double length();
  double getx();   // accessor functions
  double gety();
  double getz();
  friend ostream & operator<<(ostream & ost, V3 v);
};

ostream & operator<<(ostream & ost, V3 v);
```

We next show the implementation file `V3.cpp`, which defines the member functions. A definition of a member function `f` appearing outside the declaration of a class `C` is identical to the definition had it appeared in-line, except that the name of the function is specified as `C::f`. The constructor for class `C` will appear as `C::C`, of course. The last function in the file is the friend function `operator<<` for the class `V3`, as is customary.

```
#include <simplecpp>
#include "V3.h"

V3::V3(double p, double q, double r){  // constructor
  x = p;   y = q;   z = r;
}
```

```
                                                  // member functions
V3 V3::operator+(V3 w){ return V3(x+w.x, y+w.y, z+w.z); }
V3 V3::operator-(V3 w){ return V3(x-w.x, y-w.y, z-w.z); }

V3 V3::operator*(double t){ return V3(x*t, y*t, z*t); }

double V3::length(){ return sqrt(x*x+y*y+z*z); }

double V3::getx(){return x;}
double V3::gety(){return y;}
double V3::getz(){return z;}
                                               // other functions
ostream & operator<<(ostream & ost, V3 v){
  ost << "(" << v.x << ", "<< v.y << ", "<< v.z << ")";
  return ost;
}
```

Our file V3.cpp contained implementations of all member functions. However, it is acceptable if some of the implementations are placed in line in the header file. Typically, small member functions are left in-line in the header file, while the large member functions are moved to the implementation file.

14.8.1 Separate compilation

We can now separately compile the implementation file, and produce, for the class V3, the object module V3.o. This module, and the header file, must be given to any programmer that uses the class V3. Suppose a program using V3 is contained in the file user.cpp, then it must include the file V3.h. The program can now be compiled by specifying

```
s++ user.cpp V3.o
```

Other source/object files needed for the program must also be mentioned on the command line, of course.

14.8.2 Remarks

The general ideas and motivations behind splitting a class into a header file and an implementation file are as for functions. In whichever file the class is used, the header file must be included, because the class must be defined. The implementation file or its object module is needed for generating an executable. By not exposing the implementation file to the user of the class, we leave open the possibility that the implementation can be changed, without affecting the user program. So long as the definition in the header file does not change, the user program does not have to change.

14.9 Template classes

Like functions, we can templatize classes as well. The process of defining a class template is very similar. Here is a template version of our V3 class.

```
template<T>
class V3{
private:
  T x, y, z;
 public:
  V3(T p=0, T q=0, T r=0){ x = p;   y = q;   z = r;}
  V3 operator+(V3 w);
}

template<T>
V3 V3::operator+(V3 w){ return V3(x+w.x, y+w.y, z+w.z); }
```

The template variable T determines the type of each component x,y,z, and is expected to be specified either as float or double. We have only shown 2 member functions for brevity. One is defined in-line, the other is defined outside the class definition. Note that you must put the line template<T> before the member function defined outside as well.

Note that the template definition does not create a class, but a scheme to create a class. To create a class, you must specify a value for the template variable and affix it in angle brackets to the class-name. To create a class of the template with T being float, you simply write:

```
V3<float> a,b,c;
```

This will create the class V3<float> from the template, as well as define a,b,c to be variables of type V3<float>. In your programs, you can use V3<float> as a class name.

Note that the template for a class must be present in every source file that needs to use it. So it is customary to place it in an appropriate header file. Notice that the class is generated only when an instance is created as in the line V3<float> a,b,c; above. Thus there is no notion of separately compiling a template.

14.10 Graphics

By now you have probably realized that our graphics commands (Chapter 4 and elsewhere) are built using classes. Indeed, the names Turtle, Rectangle, Polygon, Line, Point are all names of classes. The commands to create create corresponding objects on the canvas were merely corresponding constructors. The various operations we have described on the graphics objects are member functions.

14.11 Exercises

1. Write a function which returns a circle having two given points as the endpoints of a diameter. Assume the definition of the circle structure given in Section 14.1.

2. Define the operator `>>` for the class `V3`. This should enable you to write `cin >> v;` where `v` is of type `V3`. When this is executed, the user will type in 3 floating point numbers which will get placed in `v`.

3. Define a structure for representing complex numbers. In addition to having a constructor which takes the real and imaginary parts as arguments, write a constructor which will take as arguments r, θ and returns a complex number $re^{i\theta} = r\cos\theta + i\sin\theta$. Note that this constructor cannot have just two real arguments – that will clash with the constructor taking real and imaginary parts as arguments. Add an optional argument, say a `bool` type, which if specified says whether the preceding two arguments are to be interpreted as real and imaginary parts or as $r.\theta$.

4. Define a class for storing polynomials. Assume that all your polynomials will have degree at most 100. Write a member function `value` which takes a polynomial and a real number as arguments and evaluates the polynomial at the given real number. Overload the `+,*,-` operators so that they return the sum, product and difference of polynomials. Also define a member function `read` which reads in a polynomial from the keyboard. It should ask for the degree d of the polynomial, check that $d \leq 100$, and then proceed to read in the first $d+1$ coefficients from the keyboard. Define a `print` member function which causes the polynomial to be printed. Make sure that you only print $d+1$ coefficients if the actual degree is d. Carefully decide which members will be private and which will be public. Overload the `>>, <<` operators so that the polynomial can be read or printed using them.

5. Define a structure for representing axis parallel rectangles, i.e. rectangles whose sides are parallel to the axes. An axis parallel rectangle can be represented by the coordinates of the diagonally opposite points. Write a function that takes a rectangle (axis parallel) as the first argument and a point as the second argument, and determines whether the point lies inside the rectangle. Write a function which takes a rectangle and `double` values `dx,dy` and returns a rectangle shifted by `dx,dy` in the x and y directions respectively.

6. Define a class for storing information about a book for use in a program dealing with a library. The class should store the name, author, price, a library accession number for the book, and the identification number of a library patron (if any) who has borrowed the book. This field, patron identification number could be 0 to indicate that the book is not borrowed.

   Read information about books from a file into an array of `book` objects. Then you should enable patrons to issue and return books. When a patron issues/returns a book, the patron identification number of the book should be changed. Write functions for doing this. The functions should check that the operations are valid, e.g. a book that is already recorded as borrowed is not being borrowed without first being returned.

Chapter 15

A project: cosmological simulation

It could perhaps be said that the ultimate goal of Science is to predict the future. Scientists seek to discover scientific laws so that given complete knowledge of the world at this instant, the laws will enable you to say what each object will do in the next instant. And the next instant after that. And so on. Predicting what will happen to the entire world is still very difficult, partly because we do not yet know all laws governing all objects in the world. Even if we knew all the laws, predicting what happens to a large system is difficult because of the enormous number of computations involved. However, for many systems of interest, we can very well predict how they will behave in different circumstances. For example, we understand the physics of collisions and of the materials used in a car well enough to predict how badly a car will be damaged if it collides against a barrier of certain strength at a certain velocity. The term *simulation* is often used to denote this kind predictive activity. Indeed many products are built today only after their designs are simulated on a computer to see how they hold up under in different conditions.

In this chapter and Chapter **??**, we will build a number of simulations. The simulation in this chapter is cosmological. Suppose we know the state of the stars in a galaxy at this instant. Can we say where they will be after a million years? Astronomers routinely do simulations to answer such questions. We will examine one natural idea for doing such simulations, and then examine the flaws in that idea. We will then see an improved idea, which will still be quite naive as compared to the ideas used in professional programs. We will code up this idea. We wil use our graphics machinery to show the simulation on the screen.

15.1 Mathematics of Cosmological simulation

In some sense, simulating a galaxy is rather simple. For the most part, heavenly bodies interact with each other using just Newton's laws of motion and gravitation.[1] As you might recall, the law of gravitation states that, two masses m_a, m_b with separated by a distance d attract each other with a force of magnitude

$$\frac{Gm_am_b}{d^2}$$

[1] We will stick to the non-relativistic laws for simplicity.

where G is the gravitational constant. The vector form of this is also important. If r_a, r_b are the vectors denoting the positions of the masses, then the distance between the masses is $d = |r_b - r_a|$. The force on mass m_a is in the direction $r_b - r_a$, and hence we may write the force on mass m_a in vector form as

$$\frac{Gm_a m_b(r_b - r_a)}{|r_b - r_a|^3} \tag{15.1}$$

If planets collide, then presumably more complex laws have to be brought in, which might have to deal with how their chemical constituents react. But a substantial part of the simulation only concerns how the heavenly bodies move under the effect of the gravitational force. It is worth noting that such simulations have contributed a great deal to our understanding of how the universe might have been created and in general about cosmological phenomenon. Also, the ideas used in the simulations are very general, and will apply in simulating other (more earthly!) physical phenomenon involving fluid flow, stresses and strains, circuits and so on.

Our system, then, consists of a set of heavenly bodies, which we will refer to as stars for simplicity. The state of the system will simply be the position and the velocity (magnitude and direction) of the stars. Suppose we know the initial state, i.e. for each star i we know its initial position r_i and velocity v_i (both vectors). Suppose we want to know the values after some time Δ. Letting r'_i, v'_i be the values after time Δ, we may write:

$$r'_i = r_i + \bar{v}_i \cdot \Delta$$

$$v'_i = v_i + \bar{a}_i \cdot \Delta$$

where $\bar{v}_i$ is the average velocity (vector) of the ith particle during the interval $[t_0, t_0 + \Delta]$ and $\bar{a}_i$ is the average acceleration during the interval. We do not know the average velocities and accelerations, and indeed, it is not easy to compute these quantities. However, the key observation, attributed to Euler, is that if the interval size Δ is small, then we may assume with little error that the average velocity remains unchanged during the interval for the purpose of calculating the position at the end of the interval. Euler's observation is similar to the idea we used in Section 6.6.5 to integrate $f(x) = 1/x$; the value of f was not really constant during every interval, but we assumed it is constant provided the interval is small enough. Assuming that the average velocity is simply the velocity at the beginning we may write

$$r'_i = r_i + v_i\Delta \tag{15.2}$$

Now, we can easily calculate the new position r'_i for each particle, because we know r_i, v_i. Euler's observation also applies to the acceleration: if the interval is small, then the acceleration does not change much during it. Thus the average acceleration can be assumed to be the acceleration at the beginning, and we may write:

$$v'_i = v_i + a_i\Delta \tag{15.3}$$

We are not given a_i explicitly, but we have all the data to calculate it. The acceleration of the i th star is simply the net force on it divided by its mass m_i. The net force is obtained by adding up the gravitational force on star i due to all other stars $j \neq i$. But we know how to calculate the force exerted by one star on another. Thus we may write:

$$a_i = \frac{F_i}{m_i} = \sum_{j \neq i} \frac{Gm_j(r_j - r_i)}{|r_j - r_i|^3} \tag{15.4}$$

We have described above a procedure by which we can get the state of all particles at time $t + \Delta$ given their state at time t. Our answers are approximate, but the approximation is likely to be good if Δ is small. Picking a good Δ is tricky; we will assume that we are somehow given a value for it. Suppose now that we know the state of our system at time $t = 0$, and we want the state at time $t = T$. To do this, we merely run T/Δ steps of our basic procedure! In particular, we use our basic procedure to calculate the state at time Δ given the state at time 0. Then we use the state computed for time Δ as the input to our basic procedure to get the state for time 2Δ, and so on. This may be written as:

```
1. Read in the state at time 0, i.e. the values $r_i, v_i, m_i$ for all $i$.
2. Read in $\Delta, T$.
3. For step $s = 1$ to $T/\Delta$:
    (a) Calculate $r'_i$ according to equation (15.2) for all $i$.
    (b) Calculate $a_i$ according to equation (15.4) for all $i$.
    (c) Calculate $v'_i$ according to equation (15.3), for all $i$.
    (d) Set $r_i = r'_i$, $v_i = v'_i$ for all $i$
4. end for
5. Print $r_i, v_i$ for all $i$.
```

We will not present the code for this algorithm, but you should be able to write it quite easily.

It turns out that this method can be extremely slow, because the stepsize Δ must be taken very small to ensure that the errors are small. However, there are many variations on the method which have better running time and high accuracy. One such variation employs the following rule to compute r'_i

$$r'_i = r_i + v_i\Delta + a_i\Delta^2/2 \tag{15.5}$$

where a_i is to be calculated as before. You may recognize this form. Perhaps you have studied a formula in kinematics for the case of uniform acceleration of a particle: $s = ut + at^2/2$, in which s is the distance covered, u the initial velocity, a the acceleration, and t the time. Our formula is really the same, with the acceleration, initial velocity and time being a_i, v_i, Δ respectively.

The rule to compute v'_i can also be refined as follows.

$$v'_i = v_i + \frac{a_i + a'_i}{2}\Delta \tag{15.6}$$

```
1. Read in the state at time 0, i.e. the values $r_i, v_i, m_i$ for all $i$.
2. Read in $\Delta, T$.
3. For step $s = 1$ to $T/\Delta$:
    (a) Calculate $a_i$ for all $i$ using equation 15.4, for all $i$.
    (b) Calculate $r'_i$ according to equation (15.5) for all $i$. Update $r_i = r'_i$ for all $i$.
    (c) Calculate $a'_i$ according to equation (15.4) for all $i$.
    (d) Calculate $v'_i$ according to equation (15.6). Update $v_i = v'_i$, for all $i$.
4. end for
5. Print $r_i, v_i$ for all $i$.
```

Figure 15.1: Basic Leapfrog

in which a'_i is the acceleration calculated at the new positions of the stars, i.e. using equation 15.4 but with r'_i instead of r_i. It is not hard to understand the intuition behind this formula. The acceleration at the beginning of the interval is a_i, and at the end is a'_i. The average of these, $\frac{a_i+a'_i}{2}$, is likely to be a better estimate of the acceleration during the interval rather than simply a_i. This is what the above rule uses. Equations 15.5 and 15.6 are said to constitute the *Leapfrog* method of calculating the new state. The algorithm in Figure 15.1 is based on this.

You will note that the algorithm in Figure 15.1 is inefficient: the value a'_i calculated at the end of an iteration of the loop is recalculated at the beginning of the next iteration. We will avoid this in the code we describe later.

It turns out that the Leapfrog method does indeed give more accurate results for the same value of Δ as compared to the simpler rules in Equations (15.2,15.3). Of course, the story does not end here. State of the art programs for charting the evolution of stars use even more refined methods. These are outside the scope of this book.

15.2 Overview of the program

Let us first clearly write down the specifications. Our input will be positions and velocities of a certain set of stars at time 0. We will also be given a number T. Our goal will be to find the positions and velocities of the stars at time T. We are also asked to show the trajectories traced by the stars between time 0 and time T.

The first question in writing the program is of course how to represent the different entities in the program. The main entity in the program is a *star*, of course. A star has several attributes, its velocity and position, and its mass. The mass is simply a floating point number. However, the velocity and position both have 3 components, corresponding to each spatial dimension. Clearly, we can use our `V3` class of Section 14.8 to represent positions, velocities, and accelerations. The trajectory of a star is also to be shown on the screen.

```
1. Read in the state at time 0, i.e. the values $r_i, v_i, m_i$ for all $i$.
2. Read in $\Delta, T$.
3. Calculate $a_i$ for all $i$ using equation 15.4, for all $i$.
4. Calculate $r'_i$ according to equation (15.5) for all $i$. Update $r_i = r'_i$ for all $i$.
5. For step $s = 1$ to $T/\Delta$:
    (a) Calculate $a'_i$ according to equation (15.4) for all $i$.
    (b) Calculate $v'_i$ according to equation (15.6). Update $v_i = v'_i$, for all $i$.
    (c) Update $a_i = a'_i$ for all $i$.
    (d) Calculate $r'_i$ according to equation (15.5) for all $i$. Update $r_i = r'_i$ for all $i$.
6. end for
7. Print $r_i, v_i$ for all $i$.
```

Figure 15.2: Final Leapfrog algorithm

So we probably should associate a graphics object, say a `Point`, with each star. When we compute the new position of a star, we should move the `Point` associated with the star. The star class will need a constructor and some methods to implement the position and velocity updates as per Equations (15.5,15.6).

As we mentioned in the previous section, the value a'_i calculated at the end of the sth iteration is the same as the value a_i calculated at the beginning of the $s + 1$th iteration. However, when $s = 1$, we do need to calculate a_i because there is no previous iteration. So we rearrange the code slightly, as shown in Figure 15.2.

Figure 15.2 is really a slight rearrangement of the code in Figure 15.1, in the manner of Figure 6.3. We pulled up statements 3(a), 3(b) out of the loop of Figure 15.1, and they become statements 3, 4 in Figure 15.2, and they also get added to the end of the loop, i.e. become statements 5(c), 5(d). Note that a_i of the next iteration is the a'_i of the previous, so in statement 5(c) we did not recalculate a_i, but merely set $a_i = a'_i$.

15.2.1 Main Program

The main program will create the stars. It will maintain a variable to keep track of the elapsed time. It will advance this variable in small steps to reach the given duration T. As it advances time, it will calculate the forces, and call appropriate methods on the stars to update their positions and velocities.

```
int main(int argc, char* argv[]){
  initCanvas("Star satellite system",-1,50,1000,1000);
  ifstream simDatafile(argv[1]);
  int n; simDatafile >> n;
  Star stars[n];
  const float star_radius_for_graphics = 15;

  float T, delta; simDatafile >> T >> delta;
  setup_star_data(simDatafile, stars, n, star_radius_for_graphics);

  arstep(n,stars, delta);

  for(float t=0; t<T; t+=delta){
    avrstep(n,stars, delta);
  }
  wait(5);
}
```

The program creates the canvas to show the orbits, then opens the file containing the data about the simulation. It expects the filename to be specified as a command line argument. It reads n, the number of stars, T, the time duration of the simulation, and delta the time step duration, i.e. the value Δ from the file given. Next, the function read_star_data reads the data about the stars into the array stars of class Star. It places the data read into each star object.

```
void setup_star_data(ifstream & file, Star stars[], int n, float radius){
  float mass, x, y, z, vx, vy, vz;
  for(int i=0; i<n; i++){
    file >> mass >> x >> y >> z >> vx >> vy >> vz;
    stars[i].init(mass, V3(x,y,z), V3(vx,vy,vz), radius);
  }
  assert(file);  // quick check that input was valid
}
```

Then it calls the function arstep corresponding to steps 3,4 of Figure 15.2. Then, within the loop, the function avrstep is called, corresponding to steps 5(a)–5(d).

The function arstep is as follows.

```
void arstep(int n, Star stars[], float delta){
  V3   forces[n];
  calculate_net_force(n, stars, forces);
  for(int i=0; i<n; i++)
    stars[i].arStep(delta, forces[i]);
}
```

As you can see, it calculates the forces on each star due to other stars, using the function calculate_net_force. The force on each star is passed as an argument to the arstep method of each star. The avrstep function is identical, except that it calls the avrstep method for each star.

The task of calculating forces is fairly simple as you would expect.

```
void calculate_net_force(int n, Star stars[], V3 forces[]){
  for(int i=0; i<n; i++) forces[i]=V3(0,0);

  for(int i=0; i<n-1; i++){
    for(int j=i+1; j<n; j++){
      V3 distvec = stars[j].getr() - stars[i].getr();
      double dist = distvec.length();
      double fmag = stars[i].getMass()*stars[j].getMass()/(dist*dist);

      V3 f(distvec*(fmag/dist));  // force on star i
      forces[i] = forces[i] + f;
      forces[j] = forces[j] - f;
    }
  }
}
```

Since the force due to star i on star j has the same magnitude as the force due to star j on star i, but opposite direction. So we calculate the force just once, and add it to the total force on star i, and subtract it from the total force on star j. Notice how the V3 class makes it easy to write this function.

These functions can be placed in a file, `main.cpp`.

15.3 The class `Star`

The header file `star.h` is as follows.

```
class Star {
private:
  Point visual;
  float mass;
  V3 r,v,a;  // position, velocity and previous acceleration values.
public:
  Star(){};
  V3 getr(){return r;}
  void init(float m, V3 position, V3 velocity, float radius);
  void arStep(float dT, V3 f);
  void avrStep(float dT, V3 f);
  float getMass(){ return mass;}
};
```

The data member `visual`, of class `Point`, will be used for producing the graphical animation. The `x,y` coordinates of the position (stored in member `r`) will be used as the position of each body on the screen; you may consider that we are viewing the cosmological system in the z direction, so that only the x,y coordinates are important. The member `visual` will be made to put down its pen, so that the orbit will be traced on the screen, as you will see in the member function `init`, in the implementation file `star.cpp` below.

```
#include "V3.h"
#include "star.h"
void Star::init(float m, V3 r1, V3 v1, float radius){
  mass = m;
  r = r1;
  v=v1;
  visual.init(radius,Position(0,0),Position(r.getx(),r.gety()));
  visual.setFillColor(COLOR("red"));
  visual.setFill(true);
  visual.show();
  visual.penDown();
}

void Star::arStep(float dT, V3 f){     // first step, outside loop
    a = f*(1/mass);
    V3 d = v*dT + a*dT*(dT/2);
    visual.move(d.getx(),d.gety());    // update canvas
    r = r + d;
}

void Star::avrStep(float dT, V3 f){    // basic loop step
    V3 adash = f*(1/mass);
    v = v+(a+adash)*(dT/2);
    a = adash;
    V3 d = v*dT + a*dT*(dT/2);
    visual.move(d.getx(),d.gety());    // update canvas
    r = r + d;
}
```

It should be self explanatory.

15.4 Compiling and execution

The files can be compiled by giving

```
s++ main.cpp star.cpp V3.o
```

where we assume that `V3.h` and `V3.o` from Section 14.8 are in the same directory as `main.cpp` and `star.cpp`.

To execute the program we need a file containing the data for stars. A sample file `3stars.txt` is as follows.

```
3
3000
10
100 497.00436 375.691247 0  0.466203685 0.43236573   0
```

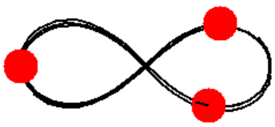


Figure 15.3: 3 stars in a figure of 8 orbit

```
100 400 400              0 -0.932407370 -0.86473146  0
100 302.99564 424.308753 0 0.466203685 0.43236573    0
```

This is meant to simulate a 3 star system for 1000 steps, with $\Delta = 10$. The initial positions and velocities of the stars are given as above. Note that they have been carefully calculated. You can simulate this system by typing:

```
./a.out 3stars.txt
```

The stars will trace an interesting figure of 8 orbit on which they will chase each other. Figure 15.3 gives a snapshot. The stars have their pen down, and hence the orbits traced are also visible.

15.5 Concluding Remarks

There are a number of noteworthy ideas presented in this chapter.

The general notion of simulating systems of interest is very important. Given the initial state of a system, and the governing laws, we can in principle determine the next states. However, as we saw, the governing laws can be applied in more or less sophisticated ways, leading to more or less error in the result. Texts on numerical analysis will indicate how the error can be estimated, and will also give even more sophisticated ideas than what we presented.

Our program also illustrates two important program design ideas. First is the idea of building classes to represent the entities important in the program. Clearly, the important entities in our program were the stars: so we built a class to represent them. But as we noted, there were many vector like entities in the problem: so it was useful to build the class `V3` as well. Finally, note that we did not write one long main program: we identified important steps in the main program and used functions to implement those steps. The functions, even if used just once, more clearly indicated the computational structure of our algorithm.

Finally, a small technical point should also be noted. We needed to create an array of `Star` objects. As we indicated in Section 14.3.1, when an array of objects is created, each object can be initialised only using the constructor which takes no arguments. Hence we had a `Star()` constructor. But this leaves open the question of how to place data in each object. For this, a common idiom is to provide an `init` member function, as we did. We call the `init` function on each object in the array and set its contents. This idiom will come in useful whenever you need arrays of objects in your programs.

15.6 Exercises

1. Build the cosmological simulation using both the methods given in the text. Use it to simulate a system consisting of a planet orbiting a star. For small enough velocities, the planet will travel around the star for both methods. You will observe, however, that for Euler's method, the orbit will keep diverging for any stepsize, which is clearly erroneous. For the same stepsize, you should be able to observe that the leapfrog orbit does not diverge, or diverges much less.

2. Consider an elastic string of length L tied at both ends. Suppose it consists of n equal weights, connected together by springs of length $L/n + 1$. Suppose each spring has Hooke's constant k, i.e. if the string is stretched by distance x, a tension $-kx$ is produced. Suppose one of the masses is moved to some new position. Suppose the string is at rest after this. Clearly, the springs on either side of the mass will stretch equally, if gravity is ignored. Now suppose the mass is released. Simulate the motion assuming there is no gravity.

3. Consider a sequence of cars travelling down a single lane road. In a simplistic model, suppose that the cars have the same maximum speed V, and acceleration a and deceleration d. Suppose each car attempts to ensure that it can come to a halt even if the car ahead of it were to stop instantaneously (e.g. because of an accident). Further assume that the driver is aware of this distance, and slows down if the distance ahead reduces, and speeds up if the distance increases, but only till the speed reaches V. Build a simulation of a convoy of cars which travels along the road on which there are signals present. When a signal turns red, the leading car in the convoy brakes so that it comes to a halt at the signal. Of course, the drivers do not react immediately, but have some response time. Note though that usually it is very easy to see if the car ahead is slowing down, because the tail red light comes on. Incorporate such details into your simulation. Show an animation of the simulation using our graphics commands.

4. Construct a class `Button` which can be used to create an on-screen button, say a rectangle, which can be clicked. Clearly, you should be able to construct buttons at whatever positions on the screen, with whatever text on them. Also, they should implement a member function `clickedP` which takes an `int` denoting the position of a click, as obtained from `getClick()`, and determine whether the click position is inside the button. What other member functions might be useful for buttons?

Chapter 16

Representing variable length entities

We continue with the idea of building classes to represent entities that we might want in our programs. In many cases, the entities we wish to represent do not have a standard length, e.g. a piece of text such as the name of a person. Indeed, human beings have names which can be very short or very long. We may also wish to represent entities such as polygons, where the number of vertices might be different, or polynomials, where the number of coefficients might be different. We may also be called upon to represent graphs (e.g. road networks) or the set of sets of students in different classes; in general we might want to represent several instances of each such collections, and the instances will typically have different lengths.

One natural strategy is to allocate the maximum possible length to represent the entity in question. We used this idea for representing text strings in Section 13.1: if we want to store names of people, we allocated `char` arrays of what we supposed was the maximum possible length. This clearly uses memory inefficiently. People have names of widely varying lengths, e.g. the actor Om Puri and the freedom fighter, scholar Chakravarti Rajagopalachari. So in general we will be forced to allocate long arrays, but most of the time we will use only small portions of these.

In this chapter, we will see how to design a data type for storing text strings, such that it does not waste memory. We cannot directly use structures/classes/arrays to represent text strings, because the size of a class/structure/array must be fixed once for all, and often without the knowledge of the size of the text string to be stored in it. The most convenient way of representing entities whose size is not known when we write the program is to use the so called *heap* memory allocation. This is also referred to as dynamic memory allocation. Using this heap memory, we will be able to construct a data type to represent text strings which will use memory efficiently.

In general the heap memory will be useful in building representations for entities whose sizes may not be known at the time of writing the program, or whose sizes may even vary over time. We will see examples of such entities in the exercises.

16.1 The Heap Memory

So far, for the most part, we have been considering variables that have been allocated in the activation frame of some function or another. Such variables are present only for the duration in which the corresponding function is executing.[1]

However, a C++ program can also be given memory outside of activation frames. A certain region of memory is reserved for this purpose. This region is called the *heap memory*, or just the heap. You can request memory from the heap by using the operator `new`. Suppose `T` is a data type such that each variable of type `T` requires s bytes of storage. Then the expression

```
new T
```

causes a variable of type `T`, or in other words s bytes of memory, to be allocated in the heap, and the expression itself evaluates to the address of the allocated variable. To use this allocated variable, you must save the address – this you can do typically by storing it in a variable of type pointer to `T`. More generally, we may write

```
new call-to-a-constructor-for-T
```

This will not just create a variable of type `T`, but it will also be initialized using the given constructor.

Thus, for the `Book` type as defined in Section 14.1, we could write:

```
Book *p;
p = new Book;
```

The first statement declares `p` to be of type pointer to `Book`. The second statement requests allocation of memory from the heap for storing a `Book` variable. The address of the allocated memory is placed in `p`. We could of course have done this in a single statement if we wish, by writing `Book *p = new Book;`. The memory allocated can be used by dereferencing the pointer `p`, i.e. we may write

```
p->price = 335.00;
p->accessionno = 12345;
```

to set the price and accession number respectively.

The second form of the new operator allows us to allocate an array in the heap. Again, if `T` is a type then we may write

```
T *q = new T[n];
```

which will allocate memory in the heap for storing an array of `n` elements of type `T`, and the address of the allocated array would be placed in `q`. We can access elements of the array starting at `q` by using the `[]` operator as discussed in Section 12.3.3. Thus we could write `q[i]` where `i` must be between 0 and `n` (exclusive). Note that `T` could be a fundamental data type, or a class. If it is a class, each object `T[i]` would be constructed by calling the constructor which does not take any arguments. You must ensure that such a constructor is available.

Note that allocating memory in this manner is a somewhat involved operation. There is some bookkeeping needed to be done so that subsequently the same memory is not allocated for another request, until we explicitly free the memory. We can free memory, i.e. return it back to the heap by using the operator `delete`. Thus, we might write:

[1] Other than this, there are the global variables. They have to be essentially allocated before the program begins execution, and hence are not interesting for the purpose of this discussion.

```
delete p;
```

Assuming p pointed to memory allocated as above, delete p; would cause the memory to be returned back, i.e. somewhere it would be noted that the memory starting at p is now free and may be allocated for future requests. The delete[] operator is used if an array was allocated. So for q as defined earlier, we may write:

```
delete[] q;
```

Note that once we execute delete (or delete[]) it is incorrect to access the corresponding address; it is almost akin to entering a house we have sold just because we know its address and perhaps have a key to it. It does not belong to us! Someone else might have moved in there, i.e. the allocator might have allocated that memory for another request. Accessing such a pointer is said to cause a *dangling pointer* error.

We used the phrase *allocator* above. By this we mean the set of (internal) functions and associated data that C++ maintains to manage heap memory. These are the functions that get called (behind the scenes, so to say) when you ask for memory using the new operator and release memory using the delete operator.

16.1.1 A detailed example

We present a detailed example of how heap memory might get allocated during execution.

Consider the program of Figure 16.1 (a). We will assume for sake of definiteness that the heap starts at address 24000. When the execution starts, all the memory in the heap is available.

When the first statement, int* intptr = new int; is executed, memory to store a single int is given from the beginning of the heap. Since an int requires 4 bytes, the 4 bytes with address 24000 to 24003 are reserved, and the address of the first of these bytes, 24000, is returned and stored in intptr. Next, memory for an array of 3 characters is requested. For this the next 3 bytes are reserved, starting at 24004. Thus cptr gets the value 24004.

The following statement *intptr = 279; stores the number 279 into the allocated memory pointed to by intptr, i.e. at address 24000. The next 3 statements store the character string constant "ab" into the array pointed to by cptr. At this stage of the execution, the memory associated with the program is in two parts: the activation frame which contains the variables intptr and cptr, and the memory which has been allocated in the heap. Figure 16.1(b) shows the activation frame. The heap is shown in Figure 16.1(c).

16.1.2 Lifetime and accessibility

We have said earlier that if a variable is created in the activation frame, then it is destroyed as soon as the control exits from the concerned function. In fact, the rule is more stringent: a variable is destroyed as soon as control leaves the block in which the variable is created.

Variables created in the heap are different. Exiting from a block, or returning from functions does not cause them to be destroyed: they can only be destroyed by executing the delete operations.

The second point concerns how the variables on the heap are accessed. They are not given a name, but are accessible only through its address! So it is vital that we do not lose

```
int main(){
  int* intptr = new int;
  char* cptr  = new char[3];
  *intptr     = 279;
  cptr[0]     = 'a';
  cptr[1]     = 'b';
  cptr[2]     = '\0';
}
```

(a)

AF of main()
intptr : 24000
cptr : 24004

(b)

Heap memory	
Address	Content
24000	279
24001	
24002	
24003	
24004	'a'
24005	'b'
24006	0
24007	
24008	
24009	
24010	
24011	
24012	. . .

(c)

Figure 16.1: (a) Program, (b) Activation Frame at end, (c) Heap area at end

the address. Thus we must not overwrite the pointer containing the address of a variable allocated in the heap, unless we stored the address in some other pointer as well. If we do overwrite a pointer containing the address of a heap variable, and there is no other copy, then we can no longer access the memory area which has been given to us. The memory area has now become completely useless. This is technically called a *memory leak*. We must not let memory leak, we must instead return it using the `delete` operator so that it can be reused!

16.2 Representing text: a preliminary implementation

We now show how to use heap allocation for representing text strings. The key idea is that the text itself will be stored in an array which we will allocate on the heap. To make the discussion more concrete, suppose we want an implementation using which we can write a main program like the following.

```
int main(){
  String a,b;
  a = "pqr";
  b = a;
  String c = a + b;  // should concatenate a, b.
  c.print();         // should print on screen

  String d[2];       // array of 2 strings
  d[0] = "xyz";
  d[1] = d[0] + c;
  d[1].print();
}
```

Other operations might also be desirable for this class, we discuss those later.

16.2.1 The basic storage ideas

Here is the declaration of a class `String` which will enable us to write the main program above.

```
class String{
  char* ptr; // will point to address in heap where actual text is stored.
public:
  String();
  void print();
  void operator=(char* rhs);
  void operator=(String rhs);
  String operator+(String rhs);
};
```

The class contains just one data member, `ptr` which is meant to point to the starting address in the heap memory where the text associated with the variable is stored. Specifically, we will store the text, terminated by a null character (i.e. `'\0'`) in the heap memory. This way, we will not need to store the length of the allocated region explicitly. Further, if a `String` variable contains the empty string, we will set its member `ptr` to `NULL`.

In principle, if two `String` variables have the same value, i.e. contain the same text, then potentially we can store a single copy of that text in the heap, and have the `ptr` members of both the variables point to that copy. While this sharing will likely save memory, it will also complicate the logic we will need to use to ask for and release heap memory. So we will adopt the simpler idea: the text associated with every variable will be stored in a distinct area in the heap memory.

16.2.2 Constructor

Initially, when we create a string variable, we want it to hold the empty string. Hence the constructor is as follows.

```
String::String(){
    ptr = NULL;
}
```

16.2.3 The `print` member function

We next discuss the member function `print`. This is very simple.

```
void String::print(){
  if(ptr != NULL) cout << ptr << endl;
  else cout << "NULL" << endl;
}
```

Since ptr gives the address from where the string is stored, it suffices to write cout << ptr << endl;. However, if ptr is NULL, then we cannot print it, instead we must explicitly print out "NULL".

16.2.4 Assignments

We discussed in Chapter 14 that assignment is already defined for structure types, provided the right hand side of the assignment is also a structure of the same type as the left hand side. Such a statement executes by copying each data member of the right hand side to the corresponding member of the left hand side. We will see that this is not adequate for our purpose. In addition, our String data type allows the right hand side to be of type char*. This we will have to define afresh. This is what we consider first.

As discussed in Section 14.3.3, in order to specify how assignment is to work, we need to define a member function operator=. Since the right hand side is to be of type char*, this member function must have a char* parameter. In the body of the function we describe what we want to happen to execute the assignment. We can define this as follows.

```
void String::operator=(char *rhs){
  delete [] ptr;
  ptr = new char[length(rhs) + 1];
  strcpy(ptr,rhs);
}
```

We give an example to see how this will work. Suppose z is of type String and say we have a statement

```
z = "mno";
```

When the function operator= executes, the implicit argument will refer to the variable z and thus ptr in the code will refer to z.ptr. Further, the parameter rhs will equal the address of the text string "mno".

Clearly, the result of the assignment should be that the string "mno" should get copied somewhere in the heap, and z.ptr should be set to point to that area of the heap.

Note that in general, the variable z may already contain some value before control arrives at the assignment statement, say the variable z contains the text "pqr". In this case, before our assignment statement, z.ptr will already be pointing to a heap region storing "pqr". When we set z.ptr to point to the area storing "mno", the area containing "pqr" will no longer be needed, and hence can be deleted. This is what the first statement in the function does. After that we request memory from the heap enough to store the new value, i.e. as many bytes as the number of characters in rhs plus an extra byte to store the null character. For this, we have used the length function from Section 13.1.4. After that we copy the text pointed to by rhs into the new region. In this we have used the function strcpy from Section 13.1.4.

Next we consider assigning one string to another as in the statement `b = a;` to execute, we need to do something much like above. The only difference is that the right hand side of the assignment is a `String` rather than a `char*`. The text that is needed to be copied now comes from taking the `ptr` member of the right hand side, rather than taking the right hand side itself directly.

```
void String::operator=(String rhs){
  delete [] ptr;
  ptr = new char[length(rhs.ptr) + 1];
  strcpy(ptr,rhs.ptr);
}
```

16.2.5 Defining operator +

Next we consider how the operation `a + b` is to be performed on `String` variables. We could write this as an ordinary function `operator+` taking two `String` arguments; or we could write it as a member function to be invoked on the left hand side `String`, with the right hand side being supplied as an argument. This function is required to return a `String` variable holding the concatenation of the text in the operands.

```
String String::operator+(String rhs){
  String res;
  res.ptr = new char[length(ptr) + length(rhs.ptr) + 1];
  strcpy(res.ptr, ptr);
  strcpy(res.ptr, rhs.ptr, length(ptr));
  return res;
}
```

The result will be calculated as the String `res`. It has to hold text of length equal to the sum of the texts of the left hand side, i.e. the text in the implicit argument, of length `length(ptr)`, and the text in the `rhs` argument, of length `length(rhs.ptr)`, and an extra byte to hold the null character. So the second statement requests memory of this size in the heap. Then the text in the implicit argument is copied into `res.ptr` using the function `strcpy`. The second `strcpy` call above assumes that there exists a `strcpy` function taking 3 arguments as follows.

```
void strcpy(char destination[], char source[], int dstart=0){
  int i;
  for(i=0; source[i] != '\0'; i++)
    destination[dstart+i]=source[i];
  destination[dstart+i]=source[i];    // copy the '\0' itself
}
```

As you can see this will copy the `source` string to the `destination` starting at index `dstart`,

This finishes the definition of the `String` class. With this the main program given in the beginning can be executed.

16.3 Advanced topics

The String class as defined in the previous section is enough for the main program given at the beginning of the section (Section 16.2). However, the definition will not allow us to do many other operations that we might want, e.g. pass String objects as arguments to functions by value, or return String objects as results. How to enable these operations is the subject of this section.

We begin by fixing a short coming of the existing definition of String. Turns out that we will have a memory leak if we allocate a String variable inside a block:

```
{
  String s;
  s = "pqr";
}
```

This is because when the block ends, the object s will be deallocated from the current activation frame. As a result we will no longer be able to use the memory pointed to be s.ptr. So ideally we should delete[] that memory at the end of every block. We could do this by writing the statement delete[] s.ptr; before the end of the block. Having to write this ourselves for every such variable and every such block is inconvenient and error prone (we might forget). But also note that ptr is a private member, so to delete it from outside the class definition we will nee some further modification to our class definition. This is where *destructor*s come in handy.

16.3.1 Destructors

Turns out that C++ allows us to specify what must happen when a class object is being deallocated, say because a block is ending. We can supply the code that we want executed as a *destructor* for the class. The destructor for a class C is named ~C, and does not take any arguments. We may specify the required delete operation inside it as follows.

```
String::~String(){ delete[] ptr; }
```

C++ will call the destructor on any variable that is about to be dellocated, and actually deallocate it only after the destructor execution finishes. Remember that the destructor call happens implicitly. So you should never explicitly call the destructor because then it will end up being called twice, with ptr being deleted twice, which is erroneous.

Note that if we do not supply a destructor, we can consider that C++ itself defines a destructor that does nothing.

16.3.2 Copy constructor

A copy constructor is a constructor, which initializes the object being created to be a copy of the argument supplied in the call. It has some special uses which we will consider; but we first note that it can be written in the natural manner.

```
String::String(const String &rhs){
  ptr = new char[length(rhs.ptr)+1];
  strcpy(ptr,rhs.ptr);
}
```

The copy constructor is essentially like the assignment operator; however since the left hand side is just being constructed, its `ptr` member does not point to anything, and hence nothing needs to be deleted.

Note that the parameter for the copy constructor must be passed by reference, the reason for this will become clear shortly.

A copy constructor can be called explicitly by the user; however there are 3 situations when it is used by the compiler.

1. When you define an object and assign another object to it at the time of definition, e.g. if you write:

   ```
   String s = "abc";
   String t = s;
   ```

   Then to copy `s` into `t` the copy constructor is used, and not the assignment operator, i.e. member function `operator=`.

2. When an object is passed to a function by value.

3. When an object is returned from a function.

Thus by defining the copy constructor yourself, you can control how the three operations above happen! If you do not define a copy constructor, C++ supplies you one, and that merely copies members. Clearly, such a default copy constructor will not be appropriate for `String`.

You will now see that it is not appropriate to pass the argument to a copy constructor by value. If you indeed pass it by value, then it would have to be first copied, but for that you would have to invoke the copy constructor, and so on. Thus we would have infinite recursion.

16.3.3 An improved assignment operator

We can improve our assignment operator slightly to allow multiple assignments in the same statement, i.e. allow us to write something like

```
String s,t,u;
s = t = u = "abc";
```

For this to happen, we merely have to return a reference to the left hand side. Thus a nicer definition of the assignment operator is as follows.

```
String& String::operator=(const String &rhs){
  delete [] ptr;
  ptr = new char[length(rhs.ptr) + 1];
  strcpy(ptr,rhs.ptr);
  return *this;
}
```

We have also passed the rhs parameter by reference.

16.3.4 Use

Figure 16.2 shows the new definitions together. We have also included member function size which gives the number of characters in the string, and the indexing operator, [].

Using this the following function and main program calling it can now be written.

```
String lcase(const String &arg){
  String res = arg;
  for(int i=0; i<res.size(); i++)
    if(res[i] >= 'A' && res[i] <= 'Z') res[i] += 'a' - 'A';
  return res;
}

int main(){
  String a,b;
  a = "PQR";
  b = a;
  String c = a + b;  // should concatenate a, b.
  c.print();         // should print on screen

  String d[2];       // array of 2 strings
  d[0] = "Xyz";
  d[1] = lcase(d[0] + c);
  d[1].print();
}
```

This will first print c, which will have the value "PQRPQR". The last print statement will concatenate "Xyz" and "PQRPQR" and then convert it all to lower case. Thus "xyzpqrpqr" will get printed.

16.4 Remarks

We have shown how we can define a data type String to store character strings. We showed that the definition was good enough to allow creating strings, indexing into them, concatenating them, assigning to strings, passing strings to functions, and returning them from

```
class String{
  char* ptr; // will point to address in heap where actual text is stored.
public:
  String(){ ptr = NULL; }
  String(const String &rhs){
    ptr = new char[length(rhs.ptr)+1];
    strcpy(ptr,rhs.ptr);
  }
  String& operator=(const char* rhs){
    delete [] ptr;
    ptr = new char[length(rhs) + 1];
    strcpy(ptr,rhs);
    return *this;
  }
  String& operator=(const String &rhs){
    delete [] ptr;
    ptr = new char[length(rhs.ptr) + 1];
    strcpy(ptr,rhs.ptr);
    return *this;
  }
  String operator+(String rhs){;
    String res;
    res.ptr = new char[length(ptr) + length(rhs.ptr) + 1];
    strcpy(res.ptr, ptr);
    strcpy(res.ptr, rhs.ptr, length(ptr));
    return res;
  }
  void print(){
    if(ptr != NULL) cout << ptr << endl;
    else cout << "NULL" << endl;
  }
  int size(){return length(ptr);}
  char& operator[](int i){return ptr[i];}
};
```

Figure 16.2: The complete `String` class

functions. Effectively, using our definition, we have an illusion that `String` is a fundamental data type. Our implementation guarantees that the objects we create will use memory efficiently.[2]

There is, however, one operation that we should not perform on the `String` class. This is the operation of allocating a `String` object itself in heap memory. Thus we should not write something like

```
Poly *ptr = new String;
```

If this is inside a block, then on exit from the block the variable `ptr` will get deallocated. As a result, the memory area it points to will leak away.

The key point is that the way the `String` class is defined, you will not need to worry about allocating memory. Indeed, you can use the `String` class without having to know about the heap. You are *not expected* to worry about managing the heap; memory will get allocated for you when needed, it will also get deallocated when needed. All this will happen behind the scenes. You are expected to sit back and enjoy the convenience, without interfering in the memory management.

16.4.1 Class invariants

While designing `String`, we made some important decisions early on. In particular we said that there will be a separate copy in the heap memory of the value stored in each `String` object. Such a property that the members of a class possess throughout their lifetime, is sometimes called a class invariant. It is useful to clearly write down such invariants, as you have seen, they guide the implementation of the class.

16.5 Exercises

1. Consider the following code. Identify all errors in it.

   ```
   int *ptr1, *ptr2, *ptr3, *ptr4;
   ptr1 = new int;
   ptr3 = new int;
   ptr4 = new int;
   ptr2 = ptr1;
   ptr3 = ptr1;
   *ptr2 = 5;
   cout << *ptr2 << *ptr1 << endl;
   delete ptr1;
   cout << *ptr3 << *ptr4 << endl;
   ```

   The possible errors are: memory leaks, dangling pointers (accessing memory that was allocated to us earlier but has since been deallocated), and referring to uninitialized variables.

[2] Except that if two variables of type `String` have the same value, we will keep two copies of the value. This can be improved upon, as discussed in Appendix C.

2. Suppose you have a file that contains some unknown number of numbers. You want to read it into memory and print it out in the sorted order. Develop an extensible array data type into which you can read in values. Basically, the real array should be on the heap, pointed to by a member of your structure. If the array becomes full, you should allocate a bigger array. Be sure to return the old unused portion back to the heap. Write copy constructors etc. so that the array will not have leaks etc.

3. Define a class for representing polynomials. Include member functions for addition, subtraction, and multiplication of polynomials.

4. Define the modulo operator `%` for polynomials. Suppose $S(x), T(x)$ are polynomials, then in the simplest definition, the remainder $S(x) \bmod T(x)$ is that polynomial $R(x)$ of degree smaller than $T(x)$ such that $S(x) = T(x)Q(x) + R(x)$ where $Q(x)$ is some polynomial.

   The main motivation for writing the modulo operator is to use it for GCD computation later. So it is important to make sure that there are no round-off errors as would happen if you divide. One way around this is to define the remainder $S(x) \bmod T(x)$ to be any $kR(x)$ where k is any number, where $R(x)$ is as defined above. Assuming that the coefficients of the polynomials are integers to begin with, you should now be able to compute a remainder polynomial without division. Hence there will be no round off either. Of course this has the drawback that the coefficients will keep getting larger. For simplicity ignore this drawback.

5. In this assignment you are to write a class using which you can represent and manipulate sets of non-negative integers. Specifically, you should have member functions which will (a) enable a set to be read from the keyboard, (b) construct the union of of two sets, (c) construct the intersection of two sets, (d) determine if a given integer is in a given set, (e) print a given set. Use an array to store the elements in the set. Do not store the same number twice. With your functions it should be possible to run the following main program.

```
main(){
  Set a,b;
  a.read();
  b.read();

  set c = union(a,b);
  set d = intersection(a,b);

  int x;
  cin >> x;

  bool both = belongs(x,d);
  bool none = !belongs(x,c);

  if( both ) {
    cout << x << " is in the intersection ";
    c.print();
  }
  else if (none) cout << x << " is in  neither set." << endl;
  else cout << x << " is in one of the sets." << endl;
}
```

The function `Set::read` will be very similar/identical to `Poly::read`. For the rest, ensure that you allocate arrays of just the right size by first determining the size of the union/intersection.

6. Euclid's GCD algorithm works for finding the GCD of polynomials as well. Write the code for this, using the iterative expression as well as the recursive expression. Will both versions cause the same number of heap memory allocations? Which one will be better if any?

7. Consider the following new member function for the class `Poly`:

   ```
   void move(Poly &dest);
   ```

   When invoked as `source.move(dest)`, it should move the polynomial contained in `source` to `dest`, and also set `source` to be undefined. Effectively, this is meant to be an assignment in which the value is not copied but it *moves*. Implement this so that the last copy rule and the distinct copy rule are respected. When the coefficients are to be moved to `dest`, is it necessary to allocate new memory?

   See if the `Poly` class with the new `move` function will improve the GCD programs considered earlier.

8. Templetize the `gcd` function so that it can work with ordinary numbers as well as polynomials. You will have to define a few more member functions as well as a constructor. Note that `int` is a constructor for the `int` type, i.e. `int(1234)` returns the integer 1234.

Chapter 17

Structural recursion

Consider the following mathematical formulae:

$$\pi = \cfrac{4}{1+\cfrac{1^2}{3+\cfrac{2^2}{5+\cfrac{3^2}{7+\cfrac{4^2}{9+\ddots}}}}}$$

and

$$\sum_{i=1}^{n} i^2 = \frac{n(n+1)(2n+1)}{6}$$

Both are correct, and the first one is rather elegant. Our concern in this chapter, however, is not the validity or elegance of these formulae. Our concern is much more mundane: how do we layout these formulae on paper. Where do we place the numerator and the denominator, how long do we make the lines denoting division? What if the denominator is itself a complicated expression as was the case in the continued fraction expansion for π? Ideally, we would merely like to somehow state the formula which we want drawn, without worrying about any geometrical aspects, and would like it if a program takes care of the rest! While this is somewhat tricky, many programs are available for doing this, the most important amongst these is perhaps the TEX program developed by Donald Knuth. The program TEX has a language for specifying mathematical formulae, and in this language, the two formulae above can be specified as:

```
\pi = \cfrac{4}{1+\cfrac{1^2} {3+\cfrac{2^2} {5+\cfrac{3^2}
{7+\cfrac{4^2} {9+\ddots}}}}}
```

and

```
\sum_{i=1}^ni^2=\frac{n(n+1)(2n+1)}{6}
```

Given this textual description, TEX can generate layouts like the ones shown. While the specification is somewhat cryptic, you can probably see some correspondence. You can

	Desired output	Input required by our program
1.	$\dfrac{a}{b+c}$	`(a/(b+c))`
2.	$a+\dfrac{b}{c}$	`(a+(b/c))`
3.	$a+b+c+d$	`(((a+b)+c)+d)`
4.	$\dfrac{x+1}{x+3}+\dfrac{x}{5}+6$	`((((x+1)/(x+3))+(x/5))+6)`

Figure 17.1: Examples of output and input

guess, perhaps, that the symbol `^` is used by TeXto denote exponentiation. Or that `\frac` and `\cfrac` somehow denote fractions. Even without precisely understanding the language of TeX you can see that the specification does not contain any geometric information. The specification does not say, for example, how long the lines in the different fractions need to be drawn. Indeed, all this is determined by TeX, using a nice blend of science and art.

How to layout mathematical formulae, is the first problem we will see in this chapter. This will turn out to be a rather interesting application of *structural recursion*. We will then go on to another important, but more classical application: using trees to maintain an ordered set in memory.

17.1 Layout of mathematical formulae

Our goal in some sense, is to write a program that does what TeX does. That, of course, is extremely ambitious! We will instead consider a very tiny version of the formula layout problem. Specifically, we will only consider formulae in which only the 2 arithmetic operators + and / are used. Our program must take any such formula, written in a language like the TeX language, and produce a layout for it. This layout must then be shown on our graphics canvas. As you will discover in the Exercises, once you master sum and division, implementing more complex operations such as other operators, summations using the $\sum$ symbol and so on, is not much more difficult. Of course, all this will still be far from what TeX accomplishes.[1]

The first question, of course, is how should we specify the input to the program. One possibility is to just use the TeX language, since that is well known. However that seems too elaborate, after all we only have 2 operators. Another possibility is to specify the formula in the style used in C++ to specify mathematical formulae. This will work, but turns out it will make it slightly harder to write our program, as you will see later. So to keep matters simple, we use a slight variation on the C++ style.

[1] TeX is a complete document processor. Furthermore, even for the purpose of laying out mathematical formulae, it is very sophisticated. For example, it adjusts sizes of the text, which our program will not.

> **Input format:** Specify the formula in the style used in C++, but place the operands to the + operator as well as the / operator in parentheses.

So as a simple example, whereas in C++ you could write `a+b`, to specify this to our program you would have to write `(a+b)`, because the rule says that the operands to every operator must be inside parentheses. Figure 17.1 gives some more examples. As you can see, the input required by our program is somewhat verbose as compared to what is required to specify the formula in C++. In the exercises we will explore the issues in allowing less verbose input.
Output requirement: Here is how the output is to be generated. When laying out sums, our program must write the summands respectively to the left and right of the + symbol. Whereas, when laying out division, we want the dividend to be above a horizontal bar, which in turn is required to be above the divisor. The divisor and the dividend must be centered with respect to the horizontal bar. Also, if a summand is a fraction, then the horizontal bar of the fraction should align with the horizontal line in the + symbol. If a summand is a simple number or an identifier, then it must align with the + symbol as in normal typing.

17.1.1 Structure of mathematical formulae

A fundamental observation is that a mathematical formula has *recursive* structure. In general a mathematical formula is built up by taking smaller mathematical formulae, and connecting them with operators. The simplest, or *primitive* mathematical formulae are plain numbers or letters, or more generally C++ style identifiers. These constitute the base cases for the recursion. As an example, the formula $\frac{a}{b+c}$ is built up by taking the two formulae a and $b+c$, and combining them using the division operator. The formula a is a primitive formula, while the formula $b+c$ is in turn built up by connecting together the primitive formulae b, c using the addition operator.

The recursive structure becomes more obvious if we first draw the formula as a rooted tree, sort of like the execution tree of Figure **??**. Figure 17.2 shows two formulae drawn as rooted trees. Leaf nodes, i.e. nodes that have no children correspond to primitive formulae. Internal nodes (i.e. nodes that are not leaves) are associated with an operator. A subtree, i.e. any node and all the nodes below it, represents a subformula used to build up the original formula. For example, in Figure 17.2(a), the subtree including and beneath the node labelled + corresponds to the subformula $b+c$. Similarly, in Figure 17.2(b), the node labelled / on the left side and the nodes below it correspond to the subformula $\frac{x+18}{x+36}$, whereas the node labelled / on the right and the nodes below it together correspond to the subformula $\frac{x}{65}$.

17.1.2 Representing mathematical formulae in a program

Figure 17.2, suggests that to represent a formula, we should first represent the nodes of the tree, and then represent the connections between the nodes, and then we should be done!

It seems natural to use a structure to represent tree nodes. The structure should contain the information associated with a node, e.g. whether the node is associated with an operator, and if so which one, or if the node is associated with a primitive formula, and if so which one. The natural way to represent *connections* between the nodes is to use *pointers*. So we define a structure `Node` as follows.

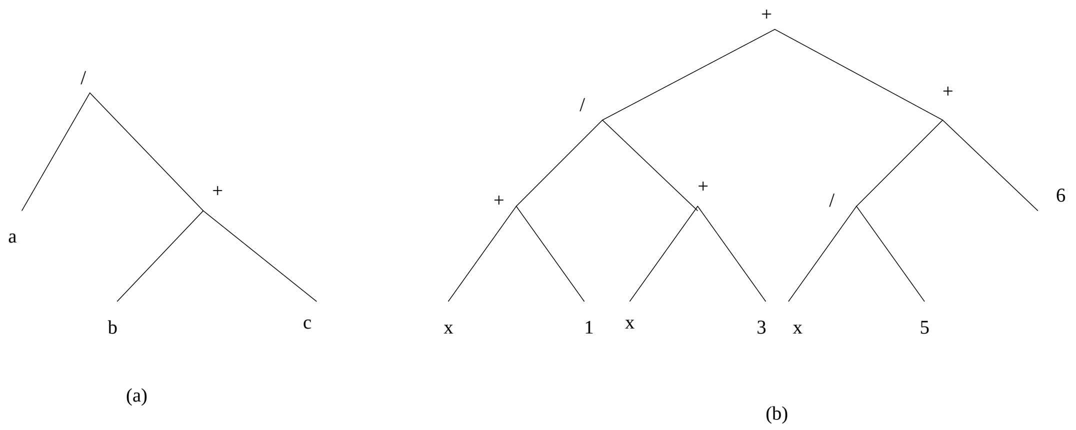


Figure 17.2: (a) Tree for $\frac{a}{b+c}$ (b) Tree for $\frac{x+1}{x+3} + \frac{x}{5} + 6$

```
struct Node{          // we give member functions later
  string value;
  char op;
  Node* lhs;
  Node* rhs;
};
```

This structure can be used to express primitive as well as non primitive formulae. If a formula is primitive, i.e. consists of an identifier or a number, we will store that symbol or number in the member `value` as a character string. Otherwise, the formula must be a binary composition of two smaller formulae. In this case, we store the operator in the `op` member, and we store pointers to the roots of the subformulae in the members `lhs,rhs`.

It is useful to have constructors for both ways of constructing formulae.

```
Node::Node(string v){                        // primitive constructor
  value = v;
  op = 'P';   // convention: 'P' in op denotes primitive formula.
  lhs = NULL;
  rhs = NULL;
}

Node::Node(char op1, Node* lhs1, Node* rhs1){  // recursive constructor
  value = "";
  op    = op1;
  lhs   = lhs1;
  rhs   = rhs1;
}
```

Here is the first way we can construct the representation for the formula $\frac{a}{b+c}$ in our program.

```
Node aexp("a");
Node bexp("b");
Node cexp("c");
Node bplusc('+', &aexp, &bexp);
  // short form for:  Node bplusc = Node('+', &aexp, &bexp);
Node f1('/', &aexp, &bplusc);
```

Thus `f1` will be the root of the tree for the formula $\frac{a}{b+c}$. Thus we can say that `f1` represents the formula. Or alternatively we can also construct a representation more directly:

```
Node f2('/', new Node("a"),
             new Node('+', new Node("b"), new Node("c"))
           );
```

An important point to note here is that the operator `new` when used on a constructor call returns a pointer to the constructed object, which is exactly what we want as an argument to our recursive constructor. Thus `f2` will also be a root of the tree for the formula $\frac{a}{b+c}$, and can thus be said to represent the formula.

There is a difference between the two constructions, however. In the first construction, all memory for the formula comes from the current activation frame. In the second construction, all memory except for the node `f2` comes from the heap, the memory for `f2` comes from the current activation frame.

Once we have a representation for formulae, our task splits into two parts:

1. Read the formula from the input in the format specified in Section 17.1 and build a representation for it using the `Node` class.

2. Generate the layout for the constructed representation. It is natural to define member functions on `Node` which will generate the layout.

We consider these steps in turn.

17.1.3 Reading in a formula

We now show how to read in the formula and build a representation. It is easiest to write our code as a constructor.

For simplicity, we will make two assumptions: (a) Each number or identifier is exactly one character long, (b) there are no spaces inside the formula. In the exercises you are asked to write code which allows longer primitive formulae and also spaces.

The idea is to read the input character by character. To read a character from a stream `infile`, we can use the operation `infile.get()` which returns the next character as an integer value. If the very first character read is a number or a letter, then we have indeed read a primitive formula and we can stop. If what is read is the character `'('`, then we know that the user is supplying us a non-primitive formula. In that case we know that we must next see in succession (a) a formula, (b) an operator, and (c) another formula. To read in (a), we merely recurse! Next we read a single character, and it had better be an operator. After that we recurse again! So our code to read in a formulae is extremely simple, even though the formulae themeselves can be very complicated.

```
  Node(istream &infile){
    char c=infile.get();

    if(c >= '0' && c <= '9' ||    // Check if it is a primitive formula.
       c >= 'a' && c <= 'z' ||
       c >= 'A' && c <= 'Z'){
      lhs=rhs=NULL; op='A'; value = c;
    }

    else if(c == '('){           // does it start a nonprimitive formula?
      lhs = node Exp(infile);       // recursively get the lhs formula
      infile.get(op);
      rhs = node Exp(infile);       // recursively get the rhs formula
      if(infile.get() != ')')
        cout << "No matching parenthesis.\n";
    }
    else cout << "Error in input.\n";
  }
```

Now we can write a part of the main program.

```
int main(){
  Node f(cin);
}
```

This will call our latest constructor with the parameter `infile` being `cin`, i.e. the formula will be read from the keyboard. You may type in something like

```
(a/(b+c))
```

and it will create `f`, just as we created `f2` earlier.

17.1.4 Drawing the formula on the canvas

The input to the drawing step is a formula, and an indication of where it is to be drawn on the canvas. It is natural that the formula is presented to us as a node `f`, which is the root of the tree representing the formula that is to be drawn. As to where to draw it, presumably the user will tell us something like "Draw the formula below and to the right of the given point (x, y)". The values x, y will be given by the user. With this we have clearly defined the specification of the problem: what we are given and what need to do. We are given a formula whose root node is `f`, and numbers x, y. We must draw f so that it lies just below and just to the right of point (x, y) on the screen.

How do we perform our task? By now you are perhaps wondering if everything with respect to trees can be done by recursion. Presumably our drawing code will also be recursive. Proceeding in this spirit, we might say that the task of drawing the formula `f` at a position (x, y) can be accomplished if we know how to draw the left hand side formula `f.lhs` at a suitable position (x', y'), and the right hand side formula `f.rhs` at a suitable position

(x'', y''), and then draw the operator or a bar between the two. We are given `f` and so we know `f.lhs`, `f.rhs` and the operator. But we do not know any of the positions. How do we figure them out?

Suppose we try out some special cases. Say `f.op` is `'+'`. Then how do we proceed? Clearly, we will need to draw `f.lhs` to the left, then the symbol `+`, and then `f.rhs`. Where exactly the symbol is to be placed depends upon the *width* required for `f.lhs`. So it would seem that we should first determine the width required. However, knowing the width is not enough – at what vertical position do we draw the `+` symbol? Perhaps we need to know the height of `f.lhs, f.rhs` as well. Continuing to try out more examples, you will perhaps realize that it is necessary but by no means sufficient to know just the width and the height of the subformulae. Consider the following examples.

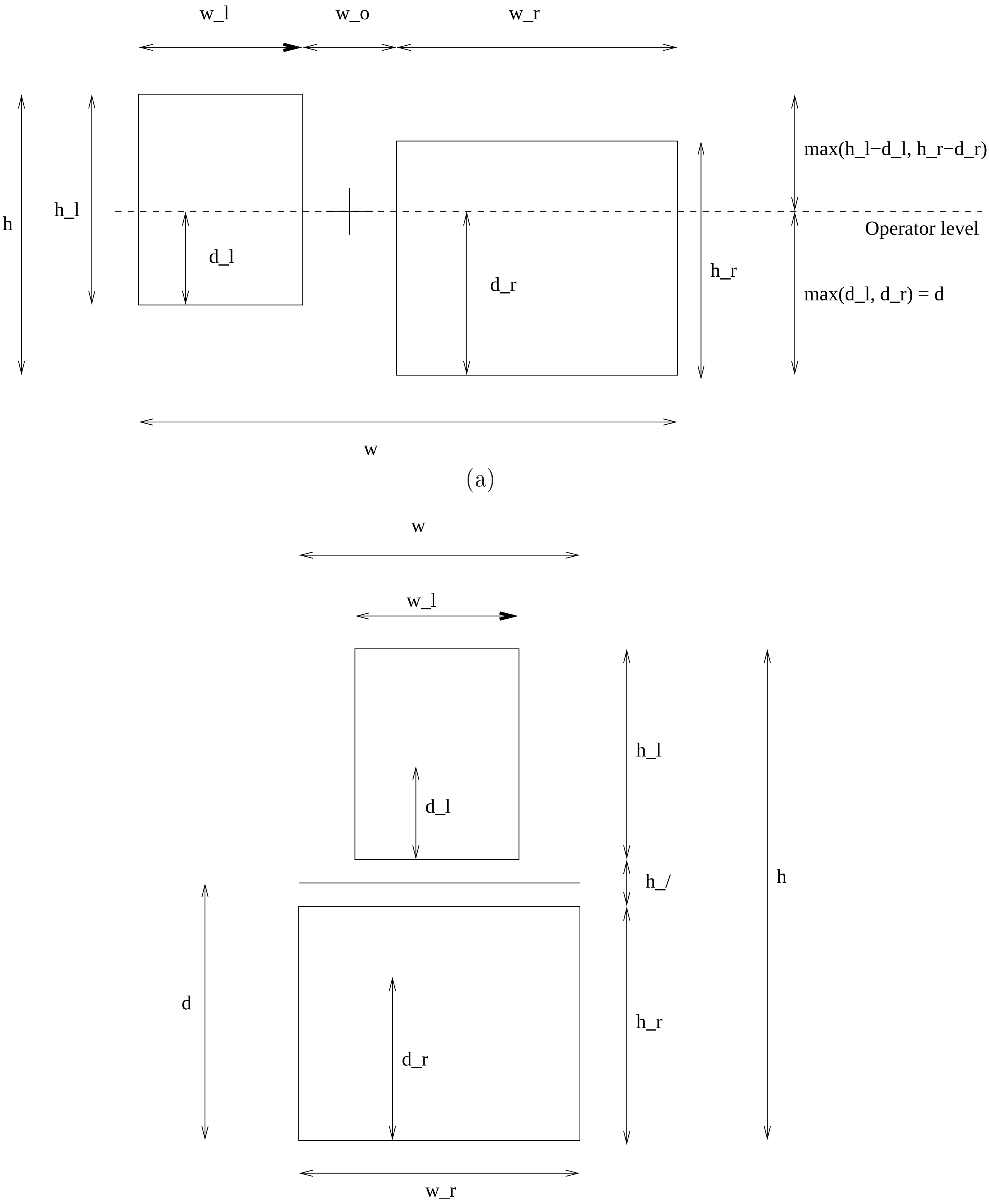
$$\frac{2}{\frac{1}{a}+\frac{1}{b}}+\frac{\frac{1}{a}+\frac{1}{b}}{2} \qquad\qquad \frac{2}{\frac{1}{a}+\frac{1}{b}}+\frac{2}{\frac{1}{a}+\frac{1}{b}}$$

The summands $\frac{2}{\frac{1}{a}+\frac{1}{b}}$, $\frac{\frac{1}{a}+\frac{1}{b}}{2}$, $\frac{2}{\frac{1}{a}+\frac{1}{b}}$ and $\frac{2}{\frac{1}{a}+\frac{1}{b}}$ all have essentially the same width and height. But their alignment in the two formulae is quite different! Clearly, we need to consider something more than just the width and the height of a subformula. Each of our summands is a fraction, and has a horizontal bar whose position seems to determine how the summands align. As was noted in the description of how the output is to be produced ("Output requirement" from Page 297), the (vertical) position of the horizontal bar is very important. This position we will call the *operator level.* Clearly, the operator level of the summands must be aligned, and the `+` connecting the summands must also be drawn aligned with the operator level.

The general situation is considered in Figure 17.3(a) for the case in which `f.op` is `+`, and in Figure 17.3(b) for the case in which `f.op` is `/`. In these figures we have shown `f.lhs,f.rhs` by their *bounding boxes.* A bounding box is simply the smallest rectangle that covers the formula. In these figures we have used w, w_l, w_r and h, h_l, h_r to respectively denote the width and heights of `f, f.lhs, f.rhs` respectively. Further define the `descent` of a formula to be the distance between the operator level and the bottom of the bounding box. The descents of `f, f.lhs, f.rhs` are respectively denoted by d, d_l, d_r. As of yet we do not know the values of the widths, heights or descents of any of the formulae. However, if we did know the values, then we can determine the mutual alignment. For example, if x, y are the coordinates of the top left corner of `f`, then then coordinates for the center of the operator `+` can be seen to be $(x + w_l + w_o/2, y + h_l - d_l)$ from Figure 17.3(a). Here w_o is used to denote the width of the operator symbol, `+` in this case, which we can determine using the function `textWidth(f.op)`. The coordinates of the top left corners of `f.lhs` and `f.rhs` will likewise also be known if we knew the widths, heights, and descents of `f.lhs, f.rhs`.

So before we can think of drawing, we must figure out the width, height and descent for each subformula in our formula. Turns out this can be done quite easily!

For the case when `op` is `+`, Figure 17.3(a) shows the relationships between the widths



(a)

(b)

Figure 17.3: Aligning layouts of `lhs` and `rhs`

w, w_l, w_h heights h, h_l, h_r and descents d, d_l, d_r: Clearly we have

$$\begin{aligned} w &= w_l + w_o + w_r, \\ h &= \max(d_l, d_h) + \max(h_l - d_l, h_r - d_r) \\ d &= \max(d_l, d_r) \end{aligned} \tag{17.1}$$

From Figure 17.3(b), we can get the relationships when `f.op` is `/`:

$$\begin{aligned} w &= \max(w_l, w_r) \\ h &= w_l + h_o + w_r \\ d &= w_r + h_o/2 \end{aligned} \tag{17.2}$$

Here we have used h_o to denote the height needed to place the horizontal bar to show the division.

The width and height of primitive formula, i.e. text, can be computed using the functions `textWidth` and `textHeight`. The descent for primitive formulae we will take as half the height. So clearly, we can compute the width, height, and descent recursively, using Equations 17.2 and 17.3. Since each formula has width, height and descent, it is most convenient to have data members `width, height, descent` in each `Node` structure. Further we will have a member function `setSize` which will calculate these numbers. So our structure becomes:

```
struct Node{
  Node *lhs, *rhs;
  char op;
  string value;
  double width, height, descent;
  Node(string v);
  Node(char op1, Node* lhs1, Node* rhs1);
  Node(istream& infile);
  void setSize();
  void draw(float clx, float cly); // to actually draw
}
```

We have already given the implementations for the constructors. The implementation for `setSize` follows the ideas described above.

```
void Node::setSize(){
    switch (op){
    case 'P':
      width = textWidth(value);
      height = textHeight(); descent = height/2;
      break;
    case '+':
      lhs->setSize();
      rhs->setSize();
      descent = max(lhs->descent, rhs->descent);
      width = lhs->width + textWidth(op) + rhs->width;
      height = descent + max(lhs->height - lhs->descent,
                             rhs->height - rhs->descent);
      break;
    case '/':
      lhs->setSize();
      rhs->setSize();
      width = max(lhs->width, rhs->width);
      height = lhs->height + Bheight + rhs->height;
      descent = rhs->height + Bheight/2;
      break;
    default: cout << "Invalid input.\n";
    }
}
```

Clearly, we have 3 cases. The first is when the expression is a primitive expression, in which case the `width,height` are merely the width and height of the `value` text string. The `descent` we have arbitrarily determined to be half the height. This should be fine tuned by looking at the picture produced by the program.

In the next case, `op` is `+` for which the operands must be drawn on either side. In this case we first call `setSize` for the `rhs` and `lhs` operands. We then determine `width, height, and ascent` as per Equations 17.2.

In the last case, `op` equals `/` for which the operands must be drawn one above the other. Again we call `setSize` on the operands, and determine the parameters as per Equations 17.3.

17.1.5 Drawing the picture

For simplicity, we will change our specification a bit. Instead of giving the coordinates (x, y) of the top left corner of the picture as an input, say we are required to draw our picture such that the left edge of the picture is at a distance `clx` from the left edge of the window, and the operator line is at a distance `cly` from the top. This would be the natural request if we are placing our drawing inline with the text that is being written. In the Exercises you are asked to modify our code that the formula is drawn so that its top left corner of the picture is at the given position.

We will define a `draw` member function taking these two numbers `clx, cly` as arguments. The implementation will of course be recursive. Since we know all sizes, we can recursively decide where each subexpression must be drawn. This is given in the code below.

```
void Node::draw(float clx, float cly){
  switch(op){
  case 'P':
    drawText(Position(clx+width/2,cly),value);
    break;
  case '+':
    lhs->draw(clx,cly);
    rhs->draw(clx+lhs->width+textWidth(op), cly);
    drawText(Position(clx+lhs->width+textWidth(op)/2,cly), string(1,op));
    break;
  case '/':
    Line(Position(clx,cly), Position(clx+width,cly)).imprint();
    lhs->draw(clx+width/2-lhs->width/2,cly-Bheight/2-lhs->descent);
    rhs->draw(clx+width/2-rhs->width/2,cly+Bheight/2+rhs->height-rhs->descent);
    break;
  default: cout << "Invalid input.\n";
  }
}
```

The simplest case is of course the base case, when the expression is primitive. In this case, we simply write the corresponding text. Remember that the function `drawtext` requires the center of the text to be specified, hence the above code calculates the x coordinate of the center by adding half the `width`, which we calculated earlier. Note that `cly`, the operator line level, directly gives the y coordinate of the center.

The code for recursive expressions is fairly natural. For the operator `+`, the operator line of the given expression is identical to the operator line of the operands. Further, the `lhs` must be aligned with the left boundary of the given expression. Hence, `lhs` must also be drawn with arguments `clx,cly`. The `rhs` has to be offset by an amount equal to the width of `lhs`, and also the width needed to draw the operator `op`. Finally we must draw `op` at its place. This is what the above code does. Note that the expression `string(1,op)` is merely a call to a `string` constructor, which returns a string containing 1 repetition of `op`.

The code for the recursive case when the operator is `/` is similar.

17.1.6 The complete `main` program

Now we can easilty complete the main program.

```
int main(){
  initCanvas("Formula drawing");
  Node f(cin);
  f.setSize();
  f.draw(200,200); // coordinates picked arbitrarily

  wait(5);

}
```

17.1.7 Remarks

Recursive structures appear in many real life situations. For example, the administrative heirarchy of an organization is recursive, e.g. there is a director/president/prime ministers, to whom report deputies, to whom report further deputies.

It is natural to associate a tree with a recursive structure. The substructures are denoted as subtrees, and the element joining the subtrees, e.g. the director will correspond to the root. In the case of mathematical expressions, the operator corresponds to the root, and the sub expressions correspond to the subtrees.

You may be wondering why we require that the formulae to be layed out be specified in our verbose format; why not just specify them as they might be in C++? It turns out that getting a program to read formulae in C++ like languages is a classical computer science problem, in its most general setting. If you pursue further education in Computer Science, you will perhaps study it in a course on compiler construction, or automata theory. For now suffice it to say that reading C++ style expressions is a difficult problem. However, in the exercises you are encouraged to think about it.

17.2 Maintaining an ordered set

We consider the following abstract problem: how to maintain a set whose elements can be integers. As the program executes, integers can get added into the set. In addition, the program must respond to *membership* queries, i.e. given some integer x, the program must determine if x is present in the set. For simplicity, we only consider these two operations, *insertion*, and *membership*. But you can see that other operations might also be useful, e.g. removing an element from a set, or finding the number of elements smaller than a given number z. Even more generally, you could let the elements of the set be complex objects, e.g. a structure containing the roll number and marks of a student. Then you might merely want to know if a student belongs to a class (membership), or you might want to know the number of students who got fewer marks than some number z. The ideas we discuss for our simple problem will be of use in these more complicated situations as well, as you will discover in the exercises.

The simplest way to store a set is to use an array, or a `vector` (Chapter 18). To add an element, we simply use the `push_back` function. To determine if an element is present, we can scan through the vector. The scanning operation, however, is rather time consuming: we need to examine every element stored in the vector. A slight improvement is to keep the elements sorted in the vector. Then we will be able to perform membership queries using binary search, which would go very fast. However, when a new element is to be inserted, we will need to find its position, and shift down the elements larger than it. This operation will on the average require us to shift half the elements, and thus is quite time consuming. So again this is unsatisfactory.

17.2.1 A search tree

There is a way to organize the elements of the set so that insertions as well as membership queries can be done very fast: we store them in a so called `search tree`. First we discuss the correspondence between a set and a search tree considered abstractly. Then we discuss how the tree can be represented on a computer.

A search tree is a rooted tree in which elements of the set are associated with each tree node. Each node has upto two outgoing branches, denoted *left*, and *right*. The left branch (if any) connects the root to the left subtree, and the right branch (if any) to the right subtree.

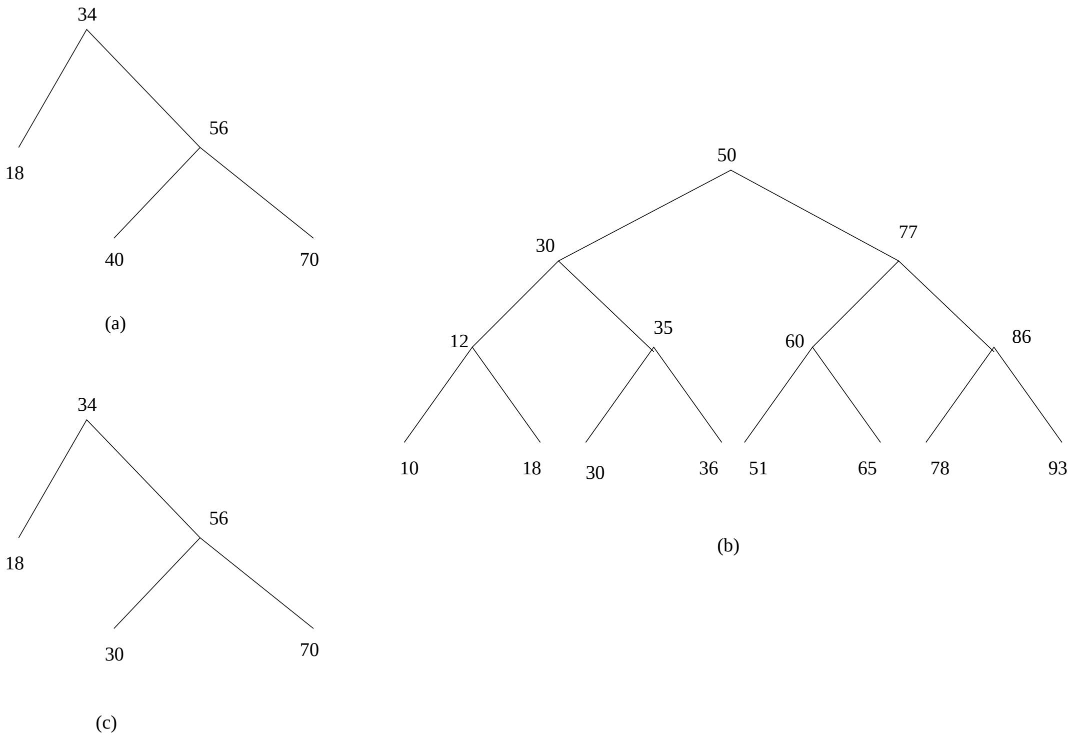


Figure 17.4: Examples (a),(b) and non-example (c) of search trees

A subtree is likewise a root with upto two branches. A subtree with no branches is called a *leaf*. Here is the key property that we require for a tree storing elements to be a search tree:

$$\begin{array}{c} \text{Values of elements in the} \\ \text{left subtree of node } v \end{array} \leq \begin{array}{c} \text{Value of element} \\ \text{at node } v \end{array} \leq \begin{array}{c} \text{Values of elements in the} \\ \text{right subtree of node } v \end{array}$$

Figure 17.4 shows example of a search tree (part (a),(b)) and a non-search tree (part (c)). In each case, we have shown the value of the element stored at each node. Thus the tree in (a) would represent the set $\{18, 34, 40, 56, 70\}$, while the one in (b) would represent $\{10, 12, 18, 30, 30, 35, 36, 50, 51, 60, 65, 77, 78, 86, 93\}$. In (c), the subtrees beneath the node with element 34 do not satisfy the search tree property: the right subtree is required to contain elements larger than 34 but it actually contains the element 30. So the tree in (c) will not represent any set, as per our scheme.

It should be clear how to represent search trees. The structure needed is almost the same as what we had for storing mathematical expressions.

```
struct Node{
  Node *lhs, *rhs;
  int value;
}
```

This is identical to what we had in Section 17.1.2, except that we do not have the member `op`, which is not needed here. We do have a member `value` which will hold the set element stored at the node. As before the members `lhs` and `rhs` will pointers to the root nodes of the left and right subtrees.

17.2.2 The general idea

We explain with an example why search trees are useful for storing sets. Suppose we have somehow constructed the search tree in memory. Suppose now the user presents us a membership query, say determine whether some number x is present in the tree. How do we do this? Remember that when we build the tree, we will only be able to refer to the root directly. To get to the other nodes, we need to follow the left or right pointers. So our problem is: we know the root of the tree that contains the elements, and we are given x, and we wish to determine if x is present at any node of the tree.

Since we only know the root of the tree, we can only compare x with the number stored at the root. If the number happens to be x, then we can immediately respond with a `true` response. If the number at the root is not x, then we need to do more work. So we ask whether x is smaller or larger than the number at the root. If x is smaller, then we know that it can only be in the left subtree. This is because of the search tree property: the numbers in the right subtree can only be larger than the number at the root, so there is no need for us to compare x to them. Thus now we can recurse on the left subtree! Similarly, if x is larger than the number at the root, then we recurse on the right subtree, i.e. try to determine if x is present in the right subtree. The recursion stops if we are forced to search an empty subtree – if that happens we know that the number is not present and we return `false`.

As an example, suppose we have in memory the tree in Figure 17.4(b). Suppose we want to know if $x = 63$ is in the tree. Then we would compare x to the number at the root, 50. Finding that x is bigger, we would decide that we only need to search the right subtree. The root node contains a pointer to the root of the right subtree, so we use that to get to root of the right subtree. So next we compare x with the number stored there, which is 77. This time we realize that x which we are looking for is smaller. So we know we must search the left subtree beneath 77. So we follow the left pointer this time and get to the node containing the key 60. This time we check and realize that x is in fact larger. So we follow the right branch out of the node containing number 60. So we get to the node containing the number 65. Since x is smaller than 65, we know we must go to the left subtree. But there is no left subtree for the node containing 65! So in this case we have determined that our number x is not present in the set. So we return `false` as the answer.

Notice that we have been able to get to an answer by examining a very few nodes: those nodes containing 50, 77, 60, and 65. We did not examine the other nodes, yet we deduced that the number $x = 63$ could not have been present in the other nodes: because we know that the tree obeys the search tree property. So this is how we can give a fast response.

If at this point you feel that our argument is too slick, you would be right. The argument given above depends very much upon the shape of the tree in which the set was stored. The tree of Figure 17.4(b) was balanced, i.e. both subtrees under each node had exactly the same number of nodes. If the tree is unbalanced, then the efficiency can become much worse. We take up this aspect in Section 17.2.5. For now, we will assume that *somehow* the trees that we encounter will be balanced, or nearly balanced. This assumption can be justified, as you will see in Section 17.2.5.

Next we consider the operation of inserting elements into the set. For this, we do something similar to checking if a number is present. Suppose we wish to insert the number x. Then we first check the number at the root. If x is smaller, then we know that it must be inserted in the left subtree, or else the search tree property will be violated. So we recursively try to insert in the left subtree. Similarly if x is larger than the number at the root, we recursively try to insert in the right subtree. The precise details will become clear when we give the code.

17.2.3 The implementation

We will indeed represent a set as a tree. In the last section, we used the root node of the tree as the representative of the tree, i.e. the point from which we can get access to the rest of the tree. This does not quite work in the present case.

There is a slight technicality that we need to consider. How do we represent an empty set? If the representation contains any `Node` object whatsoever, then that object will store a value, and hence will not represent an empty set. So clearly we cannot represent a set by the node denoting the root of the tree. Instead, we represent a set by a pointer to the node denoting the root. Now if we want to represent the empty set, we merely set this pointer to `NULL`.

For convenience, we will put the pointer to the root inside a class `Set`, which will contain a data member we will call `root`, which will point to the root node of the tree.

```
struct Set{
  Node* proot;         // pointer to tree root.
  Set(){proot = NULL;}
  // more to come.
}
```

The class will have more member functions, but we have put in a constructor which sets the member `proot` to `NULL`, indicating that the set is empty. Given this definition, we may write

```
Set mySet;    // automatically initialized to NULL, by the constructor.
```

and we will have declared a set in our program. This is nicer than saying `Node* mySet;`.

We will also change the `Node` struct slightly. Instead of using it as given earlier, we will define it as follows.

```
struct Node{
  Set lhs, rhs;
  int value;
};
```

Notice that this definition is really the same as the old, after all `Set` contains no other data members except `proot` of type `Node*`. But making the members `lhs`, `rhs` of type `Set` will make the code easier to read. Many programmers might stick with the old definition, so the Exercises ask you to do that also.

We will implement the membership query as a `bool` function `find` taking as argument the element to look for. The code for the function follows directly from what we discussed earlier.

```
bool Set::find(int elt){
  if(proot == NULL) return false;
  else{
    if(elt == proot->value) return true;
    else if(elt < proot->value) return proot->left.find(elt);
    else return proot->right.find(elt);
  }
}
```

As we said, if we reach an empty tree, the element is not present, hence the first statement returns false. Else we compare the element being searched, `elt`, with the value at the root of the set. Note however that `Set` merely contains a pointer `proot` to the root, hence the value stored at the root is `proot->value`. If `elt` is equal to `proot->value`, we have discovered that `elt` is indeed in the set, and so we return `true`. Else if `elt` is smaller than `proot->value`, we must search the left subtree, `proot->left`. Similarly the right.

Before we discuss insertion, it is useful to see a constructor for `Node`.

```
Node::Node(int v){
    value = v;
}
```

This sets the value member to the given `value`, but does nothing to the other members `lhs, rhs`. Is this OK? If nothing is specified, then these members will be initialized by their default constructors. But the default constructor for `Set` causes the member `proot` to be set to `NULL`. So the members `lhs, rhs` would indeed get initialized correctly.

Now we come to insertion. We will implement insertion by a function `insert` taking the element to be inserted as the argument. The code is recursive and follows our discussion.

```
void Set::insert(int elt){
  if(proot == NULL){proot = new node(elt);}
  else{
    if(elt == proot->value) return;        // no need to insert again.
    if(elt < proot->value) proot->left.insert(elt);
    else proot->right.insert(elt);
  }
}
```

If our set is empty, then `proot` will be `NULL`. So in this case, we will insert a node containing the new element. This is what the first statement does. If the set is not empty, then `proot` must be non `NULL`, and in this case we will come to the second line of the function. We will check if the value at the root is equal to the element being inserted, if so, we do nothing, there is no need to insert the same element again. But if the value being inserted is smaller, then we will insert into the left subtree. Similarly for the right.

The exercises ask you to define other operations, e.g. printing the set. As you might guess, most operations on trees can be naturally tackled using recursion.

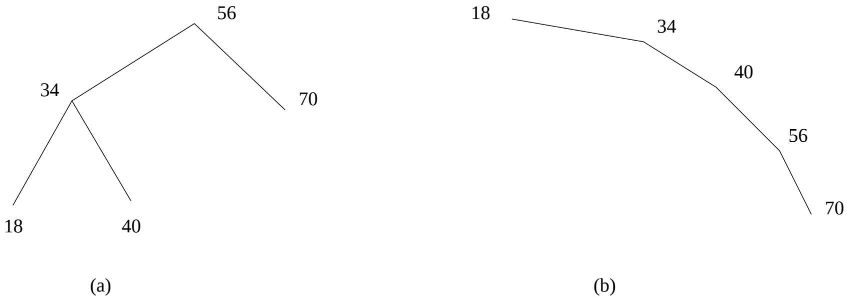


Figure 17.5: Other trees representing the same set as Figure 17.4(a)

17.2.4 A note about organizing the program

Note that our definition of `Node` refers to the definition of `Set`, and vice versa. Which one must preced the other? The definition of `Set` only refers to a pointer to `Node`. Hence we can define that after putting a forward reference to `Node`. The definition of `Node` however mentions a member of type `Set`. Hence it must come after the definition of `Set`. The implementation of the member functions in `Set` can come any place following the definition of `Set`.

17.2.5 On the efficiency of search trees

We first observe that there can be many binary search trees that contain a given set of numbers. Figure 17.5 give two additional trees which contain the same numbers as Figure 17.4(a).

There can be other trees also, in fact you should be able to prove that a set with 5 elements can be represented by 42 trees. Clearly, the time required to answer membership queries will depend upon which of these 42 trees has arisen during the execution.

Of these trees, the worst is the one in Figure 17.5(b). In this, the smallest value is at the root. The second smallest is at the right child of the root. The third smallest is at its right child, and so on. Our program will build this "tree" if the numbers were inserted in ascending order. Suppose the user asked to search for 100. Then the search would start at the root, and go rightward, examining every node in the tree. In comparison, if the set had been stored as Figure 17.5(a), then the we would first compare 100 to the value 56, and then to the value 70, and conclude that 100 is not in the set. So clearly the tree of Figure 17.5(b) is bad for the purpose of checking if 100 is in the set. Indeed, you will observe that searching in the tree of Figure 17.5(b) is essentially like searching in a sorted vector.

So perhaps we could make a general observation: our `find` function will examine the values stored in some path starting at the root and ending at the leaf. So if we want the program to run fast, then we would like all such paths to be short. It is customary to define the `height` of a tree as the length of the longest root leaf path in a tree. Thus our hope is that as the program executes, the tree we get has small height.

Theorem 4 *Suppose numbers from a certain set $|S|$ are inserted into a search tree using our* `insert` *function. Then if the order to insert is chosen at random, then the expected* *height of the tree smaller than* $2 \ln |S|$*, i.e. twice the natural log of the number of elements in the set.*

The proof of the theorem is outside the scope of this book, but the exercise asks you to validate it experimentally.

Let us try to understand what the theorem says using an example. Suppose we have a set with size 1000, whose elements are inserted in random order into our tree. Then on the average we expect to see that the height will be at most $2 \ln 1000 \leq 14$. Thus when we perform membership queries (or further insertions) we expect to compare the given number with the numbers in at most 15 nodes in the tree.

You could also ask what are the worst and best heights possible for 1000 nodes. Clearly, if the numbers came in increasing order, then we would get just one path of length 1000 – that would be the height. The other extreme is a tree in which we keep on inserting nodes as close to the root as possible. So we would start by inserting two nodes directly connected to the root, then two nodes connected to each of these, and so on, till be inserted 1000 nodes. So we would have 1 node (the root itself) at distance 0, 2 nodes at distance 1, 4 nodes at distance 2 and so on till 256 nodes at distance 8, and the remaining $1000 - 256 - 128 - \cdots - 1 = 489$ nodes at level 9. So the height of this tree would be 9.

So it is nice to know that on the average we are likely to be much closer to the best height rather than the worst. Or alternatively, on the average our `find` and `insert` functions will run fast.

17.2.6 Balancing the search tree

You might be bothered that the above program will work fast "on the average", but might take very long if you are unlucky. What if the numbers in the set got inserted in ascending order, or some such bad order?

In that case there are advanced algorithms that try to *balance* the tree as it gets built. This is done by modifying an already built tree, and say changing the root. With such rebalancing, it is indeed possible to ensure that the height of the tree remains small. Further, rebalancing algorithms have been developed that also run very fast. But this is outside the scope of this book.

17.3 Exercises

1. Extend the formula drawing program so that it allows the operators `'*'`, `'+'` and `'-'`. This is not entirely trivial: make sure your program works correctly for input `((x+3)*(x-2))`. You will see that you may need to add parenthesization to the output. For simplicity, you could parenthesize every expression when in doubt.

2. Add an operator `'^'` to denote exponentiation in the formula drawing program. In other words, `Node('^',Node("x"), Node("y"))` which will print as x^y.

3. Allow the implicit multiplication operator, i.e. it should be possible to draw $x\frac{u}{v}$.

4. Suppose the user gives the position of the top left corner of the bounding box of the formula. Show how you could do this. Also if the user asks that the formula be centered at some given point.

5. Write a constructor function which takes as input a single reference argument, `infile&` which is a reference to an `istream`, and constructs an expression based on what it reads there. The associated file should contain valid expressions but written in a *prefix* form. Note that in the prefix form, the operator comes first, and every operator is parenthesized. Thus $\frac{a}{b+c}$ will be written as (/ a (+ b c). Observe that this way of writing expressions also has a recursive structure. Thus your constructor will also be recursive. For simplicity assume that each primitive expression is a single character. Further assume for simplicity that there are no spaces in the input. Thus the above expression would appear in the file as (/a(+bc)). Note that `get` is a member function on `istreams` that can be used to read single characters. Thus `infile.get()` reads the next character from `infile` and returns its value.

6. Modify the code above so that it is allowed to contain spaces.

7. The expression `infile.peek();` returns the next character in the file without actually reading it. Use this to modify the code above so as to allow primitive expressions to be longer than a single character. Assume that consecutive primitive expressions will be separated by a space.

8. How will you represent integration with lower and upper limits, and the integrand? In other words, you should be able to draw formulae such as

$$\int_0^1 \frac{x^2 dx}{x^2+1}$$

Hint: The best way to do this is to use a ternary operator, say denoted by the letter `I`, which takes as arguments 3 formulae: the lower limit `L` , the upper limit `U`, and the expression `E` to be integrated. You could require this to be specified as `(L I U E)`.

9. As we have defined, our formulae cannot include brackets. Extend our program to allow this. You could think of brackets being a unary operator, say `B`. Since it is our convention to put the operator second, you could ask that if a formula `F` is to be bracketed, it be written as `(F B)`. Make sure that you draw the brackets of the right size.

10. You may want to think about how the program might change if the formula to be layed out is specified in the standard C++ style, i.e to draw $a + \frac{b}{c}$ the input is given as `a+b/c` rather than `(a+(b/c))` as we have been requiring. The key problem as you might realize, is that after reading the initial part `a+b` of the input, you are not sure whether the operator `+` operates on `a,b`. This is the case if the subsequent operator, if any, has the same precedence as `+`. However, if the subsequent operator has higher precedence, as in the present case, then the result of the division must be added to `a`. So you need to `look ahead` a bit to decide structure to construct. This is a somewhat hard problem, but you are encouraged to think about it. Note that your job is not only

to write the program, but also argue that it will correctly deal with all valid expressions that might be given to it.

11. Add a `deriv` member function, which should return the derivative of a formula with respect to the variable `x`. Use the standard rules of differentiation for this, i.e.

$$\frac{d(uv)}{dx} = v\frac{du}{dx} + u\frac{dv}{dx}$$

This will of course be recursive. You should be able to draw the derivatives on the canvas, of course.

12. You will notice that the result returned by `deriv` often has sub-expressions that are products in which one operand is 1 and sums in which one operand is 0. Such expressions can be simplified. Add a `simplify` member function which does this. This will also be recursive.

13. Suppose you want to represent sets using just the `Node` definition from Section17.2.1. Then to create a set `mySet` which is initially empty, I would write:

```
  Node* mySet = NULL;
```

To implement membership and insertion queries, we could merely adapt the functions `insert` and `find` of Section 17.2.3. Note however, that those were member functions for `Set`, whis is really of type `Node*`, so they cannot become member functions for `Node`. Thus they must become ordinary functions. Here is a suggested adaptation of `insert`:

```
void insert(node* set, int elt){
  if(set == NULL){set = new node(elt,NULL,NULL);}
  else{
    if(elt < set->value) insert(set->left, elt);
    else insert(set->right, elt);
  }
}
```

Do you think it is a faithful adaptation? Does it work? Hint: Be careful about whether you should use call by reference or by value.

Here is an adaptation of the find method.

```
bool find(node* set, int elt){
  if(set == NULL) return false;
  else{
    if(elt == set->value) return true;
    else if(elt <  set->value) return find(set->left, elt);
    else return find(set->right, elt);
  }
}
```

Again, is this a faithful adaptation and will it work?

14. Add a `print` member function to `Set`. Hint use recursion: first print the members in the left subtree, then the value stored at the current node, and then the value in the right subtree.

    Note: Your answer to the previous problem will likely print absolutely nothing for an empty set. Suppose that you are to print a message "Empty set" in such cases. Hint: Use one non-recursive member function which calls a recursive one.

15. Add a member function with signature `int smaller(int elt)` which returns the number of elements in the set smaller than `elt`. Hint: Add a member `count` to each node which will indicate the number of nodes in the subtree below that node. You will need to update `count` values suitably whenever you insert elements. Now use the `count` value to respond to `smaller`.

16. Experimentally verify Theorem 4. Let n denote the number of elements in the set. Assume without loss of generality that the elements in the set are integers $1, 2, \ldots, n$. Run the insertion algorithm by generating numbers between 1 and n (without replacement) in random order. Measure the height of the resulting tree. Repeat 100 times and take the average. Repeat for different values of n and plot average tree height versus n.

Chapter 18

The standard template library

An important principle in programming is to not repeat code: if a single idea is used repeatedly, write it as a function or a class, and invoke the function or instantiate a class object instead of repeating the code. But we can do even better: if some ideas are used outstandingly often, perhaps the language should *give* us the functions/classes already! This is the motivation behind the development of the standard library, which you get as a part of any C++ distribution. It is worth understanding the library, because familiarity with it will obviate the need for a lot of code which you might otherwise have written. Also, the functions and classes in the library use the best algorithms, do good memory management if needed, and have been extensively tested. Thus it is strongly recommended that you use the library whenever possible instead of developing the code yourself.

The library is fairly large, and so we will only take a small peek into it to get the flavour. In particular we will study the template classes `vector` and `map` which are among the so called *container* classes supported in the library. They can be used to hold collections of objects, just as arrays can be. Indeed you may think of these classes as more flexible, more powerful, extensions of arrays. We have hinted in Section 17.2 that arrays might not be the best way to store every collection. Indeed the SL classes such as `map` employ ideas such as balanced trees for storing elements. But of course, as users you only need to know the specification of the classes, and need not worry about how they are implemented. In addition to `vector` and `map`, we will also study the `string` class meant for storing character data. This class is extremely convenient, and you should use it by default instead of using character arrays. At the end of the chapter we will give a quick overview of the other classes in the standard library. Of these, we will use the `priority_queue` class in Chapter 21.

As running examples of the use of the standard library, we program variations on the marks display program of Section 12.2.2. Our first variation is extremely simple: the teacher enters the marks and the program must print them out in the sorted order. The main interesting feature here is that the program is not given the number of marks before hand, and so will need a flexible data structure in which to store the marks. In the second variation, the teacher also enters the roll number of each student, and we are asked to print out the marks in increasing order of the roll number or in non-increasing order of the marks. In the third variation, the teacher enters marks along with the names of the students. Then the program must wait for students to enter their names, and on receiving the name of any student, the program must print out the marks received. You know enough C++ to solve all these variations, and you have already solved some of them. However, you will see that using the Standard Library, you will be able to solve them with much less programming effort.

18.1 The template class vector

The template class `vector` is meant to be a friendlier, more general variation of one dimensional arrays. To use the template class `vector` you need to include a header file:

```
#include <vector>
```

A `vector` can be created by supplying a single template argument, the type of the elements. For example, we may create a vector of `int` and a vector of `float` by writing the following.

```
vector<int> v1;
vector<float> v2;
```

These vectors are empty as created, i.e. they contain no elements. But other constructors are available for creating vectors with a given length, and in addition, a given value. For example, you might write:

```
vector<short> v3(10);    // vector of 10 elements, each of type short.
vector<char>  v4(5,'a'); // vector of 5 elements, each set to 'a'.
vector<short> v5(v3);    // copy of v3.
```

A vector keeps track of its own size, to know the size you simply use the member function `size`. Thus

```
v3.size()
```

would evaluate to 10, assuming the definition earlier. You can access the `i`th element of a vector using the subscript notation as for arrays. For example, you could write

```
v3[6] = 34;
v4[0] = v4[0] + 1;
```

The usual rules apply, the index must be between 0 (inclusive) and the vector size (exclusive). You can also extend the vector by writing:

```
v1.push_back(37);
v3.push_back(22);
```

These statements would respectively increase the length of `v1` to 1, and of `v3` to 11. The argument to the method `push_back` would constitute the last element.

A whole bunch of operations can be performed on vectors. For example, unlike arrays, you can assign one vector to another. So if `v,w` are vectors of the same type, then we may write

```
v = w;
```

which would make v be a copy of the vector w. The old values that were contained in v are forgotten. This happens even if v,w had different lengths originally. You should realize that although the statement looks very small and simple, all the elements are copied, and hence the time taken will be roughly proportional to the length of w. You might also wonder, what happens to the memory in which the old contents of v were, specially if v was earlier much longer than w. There is a very convenient answer to this: *Dont worry about it!* The memory is managed fine. In the terminology of Chapter 16, there will be no memory leaks, no dangling pointers. The constructors, destructors, copy constructors, assignment operators of the class vector have already been written, in the style discussed in Section **??**. These will get called automatically without you having to worry that they even exist! You treat a vector like an ordinary structure; in fact you should *not* yourself use the operator new with vectors, as we indicated in Section 16.4.

You can shrink a vector by one element by writing v.pop_back(). But you can also set the size arbitrarily by writing:

```
v.resize(newSize);
w.resize(newSize,newValue);
```

The first statement would merely change the size. The second statement would change the size, and if the new size is greater, then the new elements would be assigned the given value.

18.1.1 Inserting and deleting elements

It is possible to insert and delete elements from a vector. This is discussed in Section 18.5.2

18.1.2 Index bounds checking

Instead of using subscripts [] to access elements, you can use the member function at. This will first check if the index is in the range 0 (inclusive) to array size (exclusive). If the index is outside the range, the program will halt with an error message. Note that the at function can be used on the left as well as the right hand side of assignments.

```
vector<int> v;
for(int i=0; i<10; i++) v.push_back(i*10);

v.at(0) = v.at(1);
```

This will cause the first element to be copied to the zeroth, i.e. at the end v will contain 10, 10, 20, 30, 40, 50, 60, 70, 80, 90.

18.1.3 Functions on vectors

A vector can be passed to functions by value or by reference. Because a vector is a class, if passed by value the entire vector is copied, element by element. Thus the called function gets a new copy, and the the called function can make modifications only to the copy and not the original. However, when passed by reference, the called function gets access to the original and the values in the original may be read or modified.

Here are functions to read values into a vector and print values in the vector. We have considered vectors of `int` in this example.

```
void print(vector<int> v){
  for(unsigned int i=0; i<v.size(); i++) cout << v[i] <<' ';
  cout << endl;
}
void read(vector<int> &v){
  for(unsigned int i=0; i<v.size(); i++) cin >> v[i];
}

int main(){vector<int> v(5); read(v); print(v);}
```

We may of course templatize the functions, e.g.

```
template<class T>
void print(vector<T> v){
  for(unsigned int i=0; i<v.size(); i++) cout << v[i] <<' ';
  cout << endl;
}
```

18.1.4 Multidimensional vectors

Since the template parameter in a vector names a type, by specifying that as a vector we can get a vector of vectors, i.e. equivalent of a two dimensional array.

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

This simply defines `v` to be a zero length vector of zero length vectors. Here is how we might define a length 10 vector of length 20 vectors, i.e. a 10 × 20 matrix.

```
vector<vector<int> > w(10, vector<int>(20));
```

In this we have used the two argument constructor for vectors, the first argument, 10 specifies the length, and the second element, `vector<int>(20)` gives the value of each element. But this value is itself a vector of length 20. Thus we get a 10 by 20 matrix represented.

We can access the elements of the matrix in the usual manner, i.e. by writing `w[i][j]`. However, we may also modify whole rows if we wish. Thus for `w` as defined above, we write:

```
w[0] = vector<int>(5);
```

we will change `w` to become a peculiar structure: it will have 10 rows; the first will have 5 elements, and the remaining will continue to have 20 elements.

This flexibility is very useful. Often in scientific computing, we encounter matrices of certain shapes, e.g. lower triangular matrices. In a lower triangular matrix, all elements above the main diagonal are 0. Thus we need not even store them. So we can create a vector of vectors in which the `i`th vector (`i` starting at 0) having length `i+1`. This is an easy exercise.

18.2 Sorting a vector

The standard template library contains many useful functions which you can access by including another header file.

```
#include <algorithm>
```

If you include this file, sorting a vector v is easy, you simply write:

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

That's it! This function will sort the vector v in-place, i.e. the elements in v will be rearranged so that they appear in non-increasing order. The arguments to the sort function indicate what portion of the array to sort. By writing v.begin() you have indicated that the portion to sort starts at the beginning of v, and v.end() indicates that the portion to sort ends at the end of the vector. In other words, the entire array is to be sorted. The expression v.begin() evaluates to an *iterator*. An iterator, which we will discuss in Section 18.5, is a generalization of pointers. The expression v.end() is also an iterator.

18.2.1 Marks display variation 1

We merely read the marks into a vector, and then use the sort function to sort.

```
int main(){
  vector<float> marks;

  int nextVal;
  while(cin >> nextVal)               // read into the vector marks
    marks.push_back(nextVal);

  sort(marks.begin(), marks.end());   // sort the vector

  for(int i=0; i<marks.size(); i++)   // output.  Note the use of
    cout << marks[i] << endl;         // standard array syntax.
}
```

The program above will read in all the marks (assuming the marks file is redirected to the standard input), then sort the vector, and then print out what it read.

The sort function is actually more interesting, as we will see shortly.

18.2.2 Marks display variation 2

Suppose now that our marks file contains lines of the form: roll number of the student earning the marks, followed by the marks. Again, we are not explicitly given the number of entries and the goal is to print out the list in a sorted order.

The natural way to write this program is to use a structure in which to read the roll number and marks. We would then use a vector of structures.

```
struct student{
  int rollno;
  float marks;
  bool operator<(const student& rhs) const{  // ignore this member function
    return rollno < rhs.rollno;              // for now
  }
};

int main(){
  vector<student> svec;

  student s;
  while(cin >> s.rollno){
    cin >> s.marks;
    svec.push_back(s);
  }

  // put code here to sort ******

 for(int i=0; i<svec.size(); i++)
   cout << svec[i].rollno << " " << svec[i].marks << endl;
}
```

So all that remains is the code to sort. Note that in general, it is not clear what it means to sort a vector of structures. Clearly, we must somehow specify how to compare structures, and which should be considered smaller (and hence must appear earlier in the result).

18.2.3 Customized sorting

There are two ways of doing this. The first is to define the operator `<` for comparing structs. We have given a definition of this in the code above. Basically, the definition says that when we write an expression `lhsStudent < rhsStudent`, their roll numbers will be compared, and `lhsStudent` will be considered smaller if `lhsStudent.rollno < rhsStudent.rollno`.

Given this definition, we can simply use the `sort` function as before! In other words, it suffices to replace the line marked `******` in the code with the statement:

```
sort(marks.begin(), marks.end());
```

as before. This will sort the array so that elements are arranged in non-decreasing order of `rollno`.

There is another, more general way to achieve this same effect: we somehow pass a function to `sort` which `sort` can use to compare the objects it is sorting. Perhaps the most natural way to pass a function is to pass a pointer to it, as discussed in Section 11.4. However, in this case a different mechanism is required: we must pass a so called *function object*, which we discuss next.

18.2.4 Function Objects

The first important point to note is that in C++, a function call is also an operator evaluation! Calling a function `f` with arguments `a1,a2` is simply evaluating an expression in which the operator is the function call operator denoted as `()`, and `f, a1, a2` are the operators. This way of viewing a function call might appear strange, but there is a reason for it.

So we come to the second point: we can define the behaviour of any operator for any class. This includes the function call operator `()` as well! Thus if we define `operator()` for a class `C`, and `instanceOfC` is a variable of class `C`, then we can call `instanceOfC` as a function! Here is are two examples.

```
struct C{
  bool operator()(const student &lhs, const student &rhs){
    return lhs.rollno < rhs.rollno;
  }
} instanceOfC;

struct D{
  bool operator()(const student &lhs, const student &rhs){
    return lhs.marks < rhs.marks;
  }
} instanceOfD;
```

Note that we have defined an instance of each structure as well. The function call operators in both structures expect two `student` reference arguments. Thus we can treat each structure as a function!

18.2.5 Use in the sort function

Now we can replace the line marked `******` with the line:

```
sort(marks.begin(), marks.end(), instanceOfC);
```

which will cause the array to be sorted by rollno. On the other hand, we could replace it by

```
sort(marks.begin(), marks.end(), instanceOfD);
```

which will cause the array to be sorted by `marks`. Note that in the same program you might want to first sort the array by `rollno` and then by `marks`, in which case you can make consecutive calls to `sort` by supplying `instanceOfC` first and then `instanceOfD`. Thus using the function object is more general than defining `operator<`.

18.3 The `string` class

The `string` class is a very convenient class for dealing with `char` data. It is so convenient, that you are encouraged to use the `string` class wherever possible, instead of `char` arrays. To use the string class you need to include the header file `<string>`, but note that it will be included automatically as a part of `<simplecpp>`.

We can create string objects `p,q,r` very simply.

```
#include <string>   // not necessary if simplecpp is included.

string p = "abc", q ="defg", r;
r = p;
```

The first statement will define variables `p, q, r` and initialize them respectively to `"abc"`, `"defg"` and the empty string respectively. The second statement copies string `p` to string `r`. When you make an assignment, the old value is overwritten. Notice that you do not have to worry about the length of strings, or allocate memory explicitly.

You can print strings as you might expect.

```
cout << p << "," << q << "," << r <<endl;  //prints ‘‘abc,defg,abc’’
```

This will print out the strings separated by commas. Reading in is also simple, `cin >> p;` will cause a whitespace terminated sequence of the typed characters to be read into `p`. To read in a line into a string variable `p` you can use

```
getline(cin, p);
```

Note that you cannot write `cin.getline(p)` as you might expect from your experience with `char*` variables.

The addition operator is defined to mean concatenation for strings. Thus given the previous definitions of `p,q,r` you may write

```
r = p + q;
string s = p + "123";
```

This will respectively set `r,s` to `"abcdefg"` and `"abc123"`. The operator `+=` can be used to append.

You can write `s[i]` to denote the `i`th character of string `s`. Many useful member functions are also defined. For example, the member functions `size` and `length` both return the number of characters in the string. Here are some examples.

```
s[2] = s[3];          // indexing allowed. s will become ab1123.
cout << r.substr(2)   // substring starting at 2 going to end
     << s.substr(1,3) // starting at 1 and of length 3
     << endl;         // will print out ‘‘cdefgb11’’, assuming r, s as above

int i = p.find("ab");        // find from the beginning
int j = p.find("ab",1);      // find from position 1.
cout << i << ", " << j << endl; // will print out 0, 3
```

Note that if the given string is not found, then the `find` operation returns the constant `string::npos`. We can use this as follows:

```
string t;  getline(cin, t);
int i = p.find(t);
if(i == string::npos)
  cout << "String: "<< p << " does not contain "<< t << endl;
else
  cout << "String: "<< p << " contains "<< t << " from position "<< i << endl;
```

Finally, we should note that strings have an order defined on them: the lexicographic order, i.e. the order in which the strings would appear in a dictionary. One string is considered `<` than another if it appears earlier in the lexicographical order. Thus we may write the comparison expressions `p==q` or `p<q` or `p<=q` for strings `p,q` with the natural interpretation.

18.3.1 Functions on `strings`

Since `string` is a class, we can pass it to functions using value, in which case a new copy is passed, or by reference, in which case the called function operates on the argument itself.

18.4 The `map` template class

The simplest way to think of the `map` class is as a generalization of an array or a `vector`. In an array or a vector, the index is required to be an integer between 0 and the $n-1$ if the length of the array is n. In a map, this condition is severely relaxed: you are allowed to use any value as the index, it need not even be numerical! As in an array, the value of the index determines which element of the map is being referred to.

To use the `map` template class you need to include the header `<map>`. Next, you declare the map you want.

```
map<indexType,valueType> mapname;
```

This causes a map named `mapname` to be created. It stores elements of type `valueType`, which can be accessed by supplying indices of type `indexType`. It is required that the operator `operator<` be defined for the type `indexType`. Of course, if the operator is not originally define, you can define it. However the definition should have the usual properties expected of a comparison operator, i.e. it should be transitive and asymmetric.

Let us take a simple example. Suppose we want to store the population of different countries. Then we can create a `map` named `Population`, which will store the population value (numeric). Say we store the population in billions as a unit, so our `valueType` is `double`. We would like to use the name of the country to access the element corresponding to each country, so our `indexType` could be string. So we can define our map as follows.

```
map<string,double> Population;
```

Next we insert the information we want into the map, i.e. we specify the population of different countries.

```
Population["India"] = 1.21;  // population of India is 1.21 billion
Population["China"] = 1.35;
Population["Unites States"] = 0.31;
Population["Indonesia"] = 0.24;
Population["Brazil"] = 0.19;
```

The first line, for example, creates an element whose value is 1.21, and whose index is `"India"`. You use an array access like syntax also to refer to the created elements. For example, the following

```
cout << Population["Indonesia"] << endl;
```

will print 0.24, which is the value stored in the element whose index is "Indonesia".

You have to realize that while the statements look like array accesses superficially, their implementation will of course be very different. Effectively, what gets stored when you write Population["India"] = 1.21; is the pair ("India",1.21). The name Population really refers to a collection of such pairs. Subsequently, when we write Population["India"] we are effectively saying: refer to the second element of the pair whose first element is "India". So some code will have to execute to find this element (Section 18.4.2). So a lot is happening behind the scenes when you use maps.

What if you write two assignments for the same index, e.g.

```
Population["India"] = 1.21;
Population["India"] = 1.22;
```

This will have the effect you expect: the element created the first time around will be modified so that the value stored in it will change from 1.21 to 1.22.

An important operation you might want to perform on a map is to check if the map contains an element with a given index. Suppose you have read in the name of a country into a string variable country. Say you want to print out the population of that country if it is present in the map; else you want to print out a message saying that the population of that country is not known to the program. You can write this as follows

```
cout << "Give the name of the country: ";
string country;
cin >> country;
if (Population.count(country)>0)
  cout << Population[country] << endl;
else cout << country << " not found.\n";
```

This code should immediately follow the code given above for defining the map Population and specifying the population of the various countries.

In this code the member function count takes as argument an index value, and returns 1 if an element with that index is present in the given map. Thus suppose the user typed in "India", in response to our request to give the name of a country. Then Population.count(country) would return 1 because we did enter the population of "India" into the map earlier. So in this case the final value entered, 1.22, will get printed. On the other hand, if the country typed in was "Britain", then Population.count(country) would return 0, and hence the message "Britain not found." would be printed.

You may wonder what would happen if we anyway execute

```
cout << Population["Britain"] << endl;
```

without assigning a value to Population["Britain"] earlier in the code. The execution of this statement is somewhat unintuitive. In general suppose we have defined a map

```
map<X,Y> m;
```

and suppose `x` is a value of type `X`. Then if we access `m[x]` without first assigning it a value, then implicitly this first causes the statement `m[x]=Y();` to be executed, i.e. an element is created for the index `x`, and the element stores the value `Y()` obtained by calling the default constructor of class `Y`. After that the value of `m[x]` is returned. Thus in the case of the statement `cout << Population["Britain"] << endl;`, the statement `Population["Britain"]=double();` is first executed. The constructor for the type `double` unfortunately does not initialize the value. So the map will now contain an element of unknown value but having the index `"Britain"`. Hence this unknown value would get printed.

18.4.1 Marks display variation 3

In the third variation, we had student names being entered, and after all data is entered, the program had to print marks for any student name was presented to it. Clearly, we can use `string`s to represent student names, and a map to store marks of students.

To make the problem more interesting, we will assume that for each student we have the marks in Mathematics, Physics, and Sanskrit. Further assume that the names are given in a file with lines such as the following.

```
A. A. Fair, 85, 95, 80
Vibhavari Shirurkar, 80, 90, 90
Nicolas Bourbaki, 95, 99, 75
```

i.e. the file will contain a line for each student with the name appearing first, succeeded by a comma, following which 3 numbers would respectively give the marks in the different subjects. The numbers are also separated by commas. This format, in which each line of the file contains values separated by commas, is often called the CSV format, or the “comma separated values” format.

We will use a `string` to store the student name. To store the marks, we will use a structure.

```
struct marks{
  double science, math, sanskrit;
};
```

The marks will be stored in a map, whose index will be the name of the student given as a string.

```
map<string,marks> mark_map;
```

Say our file containing the marks is named `marks.txt`. Then we can declare it in our program as

```
ifstream infile("marks.txt");  // needs #include <fstream>
```

Next we discuss how to read values from a file in the CSV format. For this we can use a form of `getline` function which allows a delimiter character to be given. The signature for this is:

```
istream& getline(istream& instream, string stringname, char delim)
```

In this, `instream` is the name of the input stream from which data is being read. The parameter `stringname` is the name of a string, and `delim` is the character that delimits the read. Thus, data is read from the stream `instream` until the character `delim` is found. The character `delim` is discarded, and the data read till then is stored into string `stringname`. Thus, we can read the name of a student by executing something like:

```
string name;
getline(infile,name,',');
```

Used with the file above, this statement will cause `name` to get the value "A. A. Fair", including the spaces inside it. Subsequently if we execute

```
getline(infile,name,',');
```

again, the string `name` would then hold the string `"85"`. Of course, we would like to convert this to a double, so we can use a stringstream:

```
double mmath;
stringstream (name) >> mmath;   // need #include <sstream>
```

As you know, this would cause the string `name` to be converted into a stringstream, from which we read into the variable `mmath`. Similarly, the other data can be read.

Figure 18.1 contains the entire program based on these ideas. In the first part, the file is read into the map `mark_map`. The first 3 values on each line, the name, the marks in math and the marks in science are comma separated. So they are used as discussed above. The last field is not comma separated, so it can be read directly. Note that when reading using the operator `>>`, the end of line character is not read. So before the next line is to be read, it must be discarded.

In the second part, the program repeatedly reads the names of students. If a name is present in the map, then the corresponding marks are printed.

18.4.2 Time to access a map

The (index,value) pairs constituting a map are stored using binary search trees like those in Section 17.2.1. The ordering rule is that all pairs in the left subtree must have index smaller than that at the root, which in turn must be smaller than the indices of the elements in the right subtree. Further, advanced techniques are used to ensure that the tree height is always logarithmic in the number of pairs stored in it. Thus making an access such as `Population[country]` happens fairly fast, i.e. in time proportional to $\log_2 n$, where n is the number of countries for which data is stored in the map.

18.5 Containers and Iterators

The classes `vector` and `map` are considered to be *container* classes, i.e. they are used to hold one or more elements. Even a `string` is thought of as a container because it contains sets of characters. There are other containers as well in the Standard Library, and we will glance at some of them shortly.

```
#include <simplecpp>
#include <fstream>
#include <sstream>
#include <map>

struct marks{
  double science, math, sanskrit;
};

int main(){
  ifstream infile("students.txt");
  map<string,marks> mark_map;

  marks m;
  string name;

  while(getline(infile,name,',')){
    string s;
    getline(infile,s,',');
    stringstream (s) >> m.math;
    getline(infile,s,',');
    stringstream (s) >> m.science;
    infile >> m.sanskrit;   // read directly, not comma terminated
    getline(infile,s);      // discard the end of the line character

    mark_map[name] = m;     // store the structure into the map
  }

  while(getline(cin,name)){
    if(mark_map.count(name)>0)
      cout << mark_map[name].math << " " << mark_map[name].science
           << " " << mark_map[name].sanskrit  << endl;
    else
      cout << "Invalid name.\n";
  }
}
```

Figure 18.1: Program for Marks display variation 3

The standard library allows some generic processing of containers, be they vectors, or maps, or even strings. For this, it is necessary to be able to refer to the elements of the container in a uniform manner. This is accomplished using an `iterator`.

An `iterator` can be thought of as a generalized pointer to an element in a container. It is intended to be used in a manner analogous to the use of an (actual) pointer in the following code which applies a function `f` to all the elements of an array.

```
int A[10]
int* Aptr
for(Aptr = A; Aptr<A+10; Aptr++)  f(*Aptr);
```

In this code we initialize the (actual) pointer `Aptr` to point to the zeroth element of `A`, and then increment it so that it points to successive elements. In each iteration we dereference it and then apply the function `f` to it. Implicit in this code is the idea that the elements are ordered in a unique manner: specifically the elements are considered in the order in which they are stored in memory.

Now we see how we can write analogous code for containers. Analogous to the actual pointer `Aptr`, we will have an iterator which will abstractly point to elements of the container, and which we can step through as the execution proceeds. In general, an iterator for a `map` can be defined as follows.

```
map<X,Y> m;
map<X,Y>::iterator mi;
```

Here `mi` is the iterator, and its type is `map<X,Y>::iterator`. Next we need to say how to set it to "point" to the first element in the map, and then how to step it through the elements. For this we first need to fix an ordering of the elements stored in the container. For `vector`s and `map`s, the elements are considered ordered according to the index, i.e. the first element is the element with the smallest index. The member function `begin` on the container returns an iterator value that abstractly point to this first element. Thus we can initialize our iterator by writing:

```
mi = m.begin();
```

An iterator supports two operations: by dereferencing you get to the element abstractly pointed to by the iterator, and by using the operator `++`, the iterator can be made to point to the next element in the order. Finally, to determine when the iterations should stop we need to know when the iterator has been incremented beyond the last element in the order. For this the member function `end` on the container is defined to abstractly point beyond the last element, just as the address `A+10` in the example above points beyond the last element of the array.

Suppose we wish to merely print all the elements in a container. Then here is how this can be done using iterators, first for the container `marks` of Section 18.2.1.

```
for(vector<float>::iterator mi = marks.begin();
      mi != marks.end();
      ++mi)
    cout << *mi << endl;
```

The code for map containers is similar. When we dereference a map iterator, we get an element of the map, which is an (index,value) pair. The pair that we get is a (template) struct, with data members first and second which hold the index and the value respectively. Since we consider an iterator to be a pointer, the struct elements can be accessed using the operator ->. Here is how we can print out the map Population of Section 18.4.

```
for(map<string,double>::iterator Pi = Population.begin();
    Pi != Population.end();
    ++Pi)
  cout << Pi->first <<": " << Pi->second << endl;
```

Similar code can be written for the string class. Note that the dereferencing operator * or the incrementation ++ should not be understood literally, these operators are given to you appropriately overloaded. But you dont need to worry about all this; you can consider iterators to be abstractions of pointers for the purpose of using them.

18.5.1 Finding and deleting map elements

Iterators are specially important for the map class. We can use the find operation on iterators to get to an (abstract) pointer to an element which has a given index value. Thus to see if the value "Britain" is stored in the map Population, we can write:

```
map<string,int>::iterator Pi = Population.find("Britain");
```

If "Britain" is not present, then Pi would take the value Population.end(). So to see if "Britain" is present and print its population we can write:

```
map<string,int>::iterator Pi = Population.find("Britain");
if(Pi != Population.end())
  cout << Pi->first << " has population "<<Pi->second << endl;
```

You can delete the element pointed to by an iterator by using the erase function as follows.

```
map<string,double>::iterator Pi = Population.find("Indonesia");
Population.erase(Pi);
```

This would remove the entry for Indonesia.

18.5.2 Inserting and deleting vector elements

Iterators can be used with vectors for inserting and deleting elements. For example, we could write

```
vector<int> v;
for(int i=0; i<10; i++) v.push_back(i*10);

vector<int>::iterator vi = v.begin()+7;
v.insert(vi,100);                       // inserting into a vector

vi = v.begin() + 5;
v.erase(vi);                            // deleting an element
```

The first two statements respectively declare a vector `v` and set it to contain the elements 0, 10, 20, 30, 40, 50, 60, 70, 80, 90. The third statement causes `vi` to point to the seventh element of `v`, i.e. the element containing 70. Then 100 is inserted at that position, the elements in the positions seventh onwards being moved down one position. The size of the vector of course increases by one. After that we set `vi` to point to the fifth element. Then that element would be deleted. This causes the subsequent elements to be moved up one position. Thus at the end the vector `v` would contain the elements 0, 10, 20, 30, 40, 60, 100, 70, 80, 90.

18.6 Other containers in the standard library

The standard library has several other containers which are very useful.

For example, the container `deque` is a double ended queue, into which you may insert or remove elements from the front as well as the back. The container `queue` allows insertions at the back and removal from the front, while the container stack requires that insertions and removals both be done from the same end.

An important container is the priority queue. You can insert elements arbitrarily, however, when removing elements, you always get the *smallest* element inserted till then. We discuss and use priority queues in Chapter 21.

An interesting container is the `set`. This supports operations for inserting elements and subsequently finding them. The elements are required to have `operator<` defined on them, and this order is used for storing the elements in a binary search tree, just as a map was stored in a binary search tree. The elements are ordered according to the `operator<` order defined on them, and will get printed in this order if printed using iterators as in Section 18.5. So if we store elements in a `set`, we really dont need to explicitly sort them.

This description is of course very sketchy. You should consult various standard library references on the web to get details.

18.7 Exercises

1. Explain what each statement of the following code fragment does.

```
vector<int> a(5,33);
vector<char*> countries(4);
vector<vector<double> > v(3,vector<double>(5, 3.14));
```

2. Write a code fragment that creates a 10 × 10 matrix stored as vector of vectors of doubles and initializes it to the identity matrix.

3. Write a program to multiply two matrices of arbitrary sizes represented as vector of vectors.

4. Write a function which returns a lower triangular matrix using a vector of vectors. Specifically, you should only allocate space to store elements a_{ij} where $j \leq i$.

5. Define a class `LTM` for storing lower triangular matrices, with signature as follows.

   ```
   class LTM{
     vector<vector<double> > data;
   public:
     LTM(int n);
     double getElem(int i, int j);
     void setElem(int i, int j, double v);
   }
   ```

   As you might guess the constructor constructs an LTM matrix with the given number of rows and columns. The member functions return the element at index `i,j` and assign the value `v` to the element at index `i,j` respectively. Note that if `j>i` then `getElem` must return 0. If `j>i` the `setElem` must do nothing and print a message. Give implementations of all the member functions.

6. Write a program that will receive information about the states of India and their capitals and answer questions about these when asked. Specifically it should process 3 kinds of commands. The first kind is:

   ```
   Learn state capital
   ```

   As an example, the user may type `Learn Maharashtra Mumbai`. In this case this information must be remembered by the program. The second kind of command is

   ```
   Tell capital-or-state-name
   ```

   For example, the user may type `Tell Gandhinagar`, whereupon the program must respond that it is the capital of Gujarat. Likewise if the state is given its capital must be given in response. The third kind of command is just

   ```
   Exit
   ```

   whereupon the program must exit. Use vectors and strings.

7. Write a program that will receive information about towns and the states in which they are located and answer questions about these when asked. Specifically it should process 3 kinds of commands. The first kind is:

   ```
   Learn state town
   ```

   As an example, the user may type `Learn Maharashtra Pune`. In this case this information must be remembered by the program. The second kind of command is

   ```
   Describe state
   ```

For example, the user may type `Describe Karnataka`, whereupon the program must respond that with all the towns it knows in `Karnataka`. The third kind of command is just

```
Exit
```

whereupon the program must exit. Use vectors and strings.

8. Write a program that prints out all positions of the occurrences of one string `pattern` inside another string `text`.

9. You are to design a class to store sparse polynomials i.e. polynomials in which even if the degree of a polynomial is n, there may be far fewer terms in the polynomial, i.e. many of the powers might have coefficient 0. In such case, it may be wasteful to allocate an array or vector of size $n + 1$ to store a polynomial. Instead, it might be more efficient to store only the non-zero coefficients, i.e. store the pair (i, a_i) if the coefficient a_i of x^i is non-zero. Use a `map` to store such pairs. Write functions to add and multiply polynomials. Note that iterators on maps will go through stored pairs in lexicographical order. Exploit this order to get efficient implementations.

10. Suppose for each student we know the marks in several subjects. The total number of subjects might be very large, of which each student might have studied and got marks in some. Write a program which reads in the marks a student has obtained in different subjects, and then prints out the marks obtained given the name of a student and the name of the subject for which the marks are requested.

    You are expected to use a `map` to store the data for all students, and a map for each student in which to store the marks for the different subjects taken by the student.

11. The algorithm collection in standard library also contains a `binary_search` function for performing binary search on sorted containers such as vectors. The signature of this function is

    ```
    bool binary_search(ForwardIterator first, forwardIterator last,
                       const T& value_to_search);
    ```

    Here the region of the container between `first` (inclusive) and `last` (exclusive) is searched to find an element equal to `value_to_search`. The type of the element stored in the container must be `T`. An additional argument, a functional object is also allowed. The functional object must effectively behave like the `<` operator. Use this to implement variation 3 of the marks display program, using just vectors rather than `maps`.

    The function `binary_search` is guaranteed to execute in logarithmic time when used with `vector` containers.

Chapter 19

Inheritance

Inheritance is one of the most important notions in object oriented programming. The key idea is: you can create a class `B` by designating it to be *derived* from another class `A`. Created this way, the class `B` gets ("inherits") all the data members and ordinary function members of `A`. The designation process does not require any code from `A` to be copied to `B`, in fact, avoiding multiple copies of code is an important motivation for inheritance. Of course, `B` need not be an exact replica of `A`, and the programmer may give additional members to `B`. Furthermore, the programmer may change some of the inherited function members. As you might suspect, this is a convenient way to create new classes. The most common (and recommended!) use of inheritance is in the following setting. Suppose class `A` represents a real-life entity A. Suppose another real-life entity B is a specialized version of A. Then B can often be represented using a class `B` derived from class `A`.

It is customary to say that class `B` is a *subclass* of class `A`, is *derived* from `A`, or obtained by *extending* `A`. And of course, `B` is said to *inherit* from `A`. It is likewise customary to say that `A` is a *superclass* of `B`, or *base* class of `B` or sometimes the *parent* class of `B`. An important point to be noted is that the process of inheritance does not change the superclass in any way.[1] We can have several classes say `B,C,D` inheriting from `A`, and perhaps classes `E,F` inheriting from `B`. In such a case, the classes `A, B, C, D, E, F` are said to constitute an *inheritance heirarchy.*

Inheritance can play a central role in the design of complex programs. Often, a program deals with categories of objects, which are divided into subcategories. For example, a program might be concerned with bank accounts, and these may be divided into different types of accounts, e.g. *savings* accounts and *current* accounts. In such cases it turns out to be useful to represent a category (e.g. accounts) by a class, and subcategories (savings accounts and current accounts) by subclasses. We will see an example of this idea in Section 19.4, and more substantial examples in the next chapter.

In this chapter we are concerned more with the mechanics of inheritance. So we begin by considering a few simple examples. The goal in these examples is to merely create a new class `B` which is similar but not identical to a given class `A`. We will inherit the common part and separately define in `B` the part that is unique to `B`. Of course, this requires `A` to be written in a manner that will facilitate the inheritance. But with some foresight, we can design classes so that it is possible to create subclasses from them later.

[1] This is as in real life: inheritance affects the children but not the parents.

19.1 Turtles with an odometer

In our first example, we will design a class meteredTurtle which is exactly like the class Turtle, except that the turtle will keep a count of how much distance it has covered. So a meteredTurtle will be able to move forward, turn, change colours etc. just like a Turtle, but in addition it will have an additional member function distanceCovered which will return the total distance covered till then. Here is how we can define meteredTurtle.

```
class meteredTurtle : public Turtle{
  float distance;
public:
  meteredTurtle(){
    distance = 0;
  }
  void forward(float d){
    distance += abs(d);   //
    Turtle::forward(d);
  }
  float distanceCovered(){
    return distance;
  }
};
```

The first line of the definition has a new part ": public Turtle". This says that the class meteredTurtle is being defined by inheriting from the class Turtle, with the type of inheritance being public. Note that in order to use the class Turtle in this manner, it should be first defined. This can be done by having the line #include <simplecpp> first, which we have not shown. C++ allows several types of inheritance; the most common of these is public. We will discuss other kinds of inheritance later, but in this book unless specified otherwise we mean public inheritance when we speak of inheritance.

The body of the definition merely states *what is different in the subclass* meteredTurtle *as compared to the superclass* Turtle. Thus the first line inside our meteredTurtle definition defines a private member distance. This will be an additional data member that every meteredTurtle (instance) has, over and above all the members that Turtle (instances) have. The definition of meteredTurtle also contains three member function definitions. These will also be available for use on instances of meteredTurtle. The first member function is a constructor for the new class. Note that constructors and destructors are not inherited, so you must define them afresh for each subclass.

The constructor for meteredTurtle merely initializes the new member, distance, to 0. This is meant to signify that at the beginning the turtle has not travelled any distance. You may wonder how the inherited members of meteredTurtle get initialized. Indeed, as we will see later, the class Turtle contains many data members which are used to hold information about the turtle such as its position on the screen, colour, and so on. These get initialized because of the following rule of C++: before executing the code for a constructor of a class, initialize the inherited members by calling the constructor of its superclass. Thus before setting distance to 0, a call is implicitly made to the default constructor of Turtle, which causes the inherited members to be initialized. Notice that this is very convenient: as a programmer you do not even need to know what the class is inheriting, and you can still be assured that the inherited members will be initialized!

Next we have a definition of the member function `forward`. Note first that `Turtle` already has a `forward` member function. So if we define `forward` again, then it means that for `meteredTurtle` objects, the new definition is to be used rather than the one that would have been inherited from `Turtle`. In the new definition, the value by which we move forward is first added to `distance`. Since the argument to `forward` can be negative, we take the absolute value. The last statement in this function is noteworthy: `Turtle::forward(d)`. This calls the `forward` function from the `Turtle` class. The `forward` function from the `Turtle` class causes the turtle to move forward on the screen. So this is what the last statement does.

The last function `distanceCovered()` is new, and not present in `Turtle`. This function can be used only with instances of `meteredTurtle` and not with instances of `Turtle`. As you see, this function merely returns the value of the data member `distance`.

Here is a main program that uses the above definition.

```
main_program{
  initCanvas("Metered turtle");
  meteredTurtle mt;

  mt.forward(100);
  mt.left(90);
  mt.forward(100);
  mt.left(90);
  mt.Turtle::forward(100);
  cout << mt.distanceCovered() << endl;
}
```

We create the metered turtle `mt` and then move it forward by 100 pixels 3 times, turning right 90 degrees between the moves. For the first two moves our code uses just `forward`, as a result the new `forward` function gets used. This will move `mt` forward, and will also increment the `distance` member in `mt`. The third time we have used `Turtle::forward`, this causes the method from the `Turtle` class to be used. Note that this will not increment `distance`. Hence, by the time control arrives at the last statement, `distance` will have got increased to only 200, which will get printed. Of course, in practice you are expected to use only `forward` with `meteredTurtle` and not `Turtle::forward`, if you want the `distanceCovered` function to correctly report the distance covered.

19.2 Another example

The second example we consider is for the `V3` class as defined in Section 14.8. Suppose we wish to add a function to this class which computes the vector dot product. Remember, given two vectors (a, b, c) and (d, e, f) their dot product is the number $ad + be + cf$. You might ask why not just modify the `V3` class code and add our new member function. Suppose you do not have the source program, you have only been given the object module and the header file. In this case, inheritance provides an elegant way to extend the V3 class. You simply define a class myV3 which inherits from V3, and into that you add the new dot product function. The myV3 class will inherit all other functions and will also be able to perform dot products. Here is a definition of myV3 and a small program to test it.

```
#include "V3.h"
class myV3 : public V3{
public:
  myV3(double p=0, double q=0, double r=0) : V3(p,q,r){ // constructor
  }
  double operator*(myV3 &v){                             // dot product
    return getx()*v.getx() + gety()*v.gety() + getz()*v.getz();
  }
};

int main(){
  myV3 X(1,2,3), Y(4,5,6);
  cout <<X.length() << endl;
  cout << X*Y << endl;
}
```

Our new class does not have any new data members, but just a constructor and the function to compute the dot product.

We noted in the previous section that the constructor of the superclass is implicitly executed to initialize the inherited members before the before the constructor of the subclass is executed. So in this case, a constructor of the V3 class would get executed. It is possible to specify which V3 constructor to use, and what arguments to use for the call. For this we simply write the call to the constructor after the parameter list and before the body, following a ":". Thus in our example, the text ": V3(p,q,r)" does this. Thus the inherited members are initialized using the call V3(p,q,r). This causes the inherited members (Section 14.8) x,y,z respectively to get the values of p,q,r. We will see this form in more detail in the next section.

Then we have the definition of the dot product as the operator *. Note that this uses the *accessor* functions getx, gety, getz rather than directly access the data members x,y,z. The reason for this will also be seen in the next section.

The main program will first print the length of X, $\sqrt{1^2 + 2^2 + 3^2} = \sqrt{14}$, using the inherited function length. Then it will print the dot product, $1 \times 4 + 2 \times 5 + 3 \times 6 = 32$, using the new member function operator*.

19.3 General principles

In general we can define a class B can as a subclass of a class A, by writing:

```
class B : public A {
  // how B is different from A
}
```

This will create a class `B` which will have all data members and ordinary function members of `A`. Indeed, we can assume that an object of class `B` will contain inside it an object of class `A`. We will call this inner object the *inherited* object. The body of the definition will describe additional data members, and additional member functions that `B` has. The body may contain redefinitions of inherited member functions as well. For example, suppose the body contains a definition of `f`, which is an inherited member function, i.e. a function already defined in `A`. In such a case, the new definition is to be used with instances of `B`. The new definition is said to *override* the old definition, and will be used for objects of class `B`. The definition of `f` from `A` will continue to be used for objects of class `A`. Instances of class `B` can also use the old definition from `A` if necessary. Only, to do that, a slightly involved syntax is needed. Instead of just using the name `f` of the function, we must write `A::f`.

Note that the class `A` must itself be defined when we define `B`. This is typically accomplised by including the header file of `A`.

Although the each object of class `B` contains inside it an object of class `A`, i.e. the inherited object, the definition of `B` may not have access to all members of `A`. How data and function members of `A` can be be used in class `B` is determined by the following rules.

19.3.1 Access rules and `protected` members

Suppose that `m` is a data or function member of `A` and `b` is an instance of `B`. Then the manner in which the member `m` of instance `b` can be accessed is determined by the following rules.

Case 1: `m` is a public member of `A`: In this case, we can consider `m` to be a public member of `B` as well. In other words, `m` can be accessed inside the definition of `B`, as well as outside the definition of `B`.

Case 2: `m` is a private member of `A`: Then `m` cannot be accessed even inside the definition of `B`. And of course it cannot be accessed outside. In other words, private members are accessible only inside the definition of the class in which the member was defined (and its friend classes). The subclass instances cannot directly access private members of the superclass. This is not to say that private members of the superclass are useless. There might well be a public or protected (see below) member function `f` of `A` which refers to `m`. Now, the code in `B` can refer to `f`, and hence it will indirectly refer to `m`. We saw an example of this when we used accessor functions `getx, gety, getz` to access private members `x,y,z` of `V3` while defining `myV3` in Section 19.2.

Case 3: `m` is a "protected" member of A: The notion of `protected` is as follows. If a member of a class `A` is designated as `protected` then it can be accessed only inside the definition of `A` or of its subclasses. In other words, it is less accessible than a public member (which is accessible everywhere), and more accessible than a private member, (which is accessible only in the definition of `A`).

Here is a code snippet that gives an example of the above rules.

```
class A{
```

```
private:
  int p;
protected:
  int q;
  int getp(){return p;}
public:
  int r;
  void init(){p=1; q=2; r=3;}
};

class B: public A{
public:
  void print(){
    cout << p << endl;  // compiler error.  p is private.
    cout << q << endl;
    cout << r << endl;
    cout << getp() << endl;
  }
};

int main(){
  B b;
  b.init();
  cout << b.p         // compiler error.  p is private
       << b.q         // compiler error.  q is protected
       << b.r
       << b.getp()    // compiler error.  getp is protected.
       << endl;
}
```

If you compile this code, (with `#include <simplecpp>` at the top), you will get the 4 compiler errors as marked. Compiler errors 1 and 2 are because `p` is private in `A`, and can hence not be accessed in the definition of `B`, or in `main`. Compiler errors 3 and 4 are because `q` and `getp` are protected in `A`, and hence cannot be used outside of the definition of `A` or of any subclass of `A`. Indeed you will see that protected members `q` and `getp()` can be used fine inside the definition of `B`. Further, the public members `init`, and `r` can be used if needed in both `B` as well as `main`.

Once the offending parts are removed, you will see that the code will compile fine, and print 3 as a result of the print statement in `main`.

19.3.2 Constructors and destructors

Suppose that class `B` is a subclass of class `A`. A constructor for `B` has the following general form.

```
  B(constructor arguments) : call-to-constructor-of-A
```

```
    {
     // code which constructs new members of B
    }
```

The `call-to-constructor-of-A` merely constructs the inherited object (of class `A`) contained inside the instance of `B` being created. An instance of `B` also contains new members, these are constructed inside the body of the constructor. The part

```
                              : call-to-constructor-of-A
```

is optional. If it is omitted, the default constructor of `A` gets called. The constructor of `A` is called before the body of the constructor of `B` is executed. This is very similar to what happens for nested structures (Section 14.4.4).

In Section 19.1 you saw an example in which the default constructor of `Turtle` got used for creating a `meteredTurtle`. Suppose now that we had an alternate constructor for `Turtle` which took arguments `x,y` giving the initial position for the turtle. Then we could write an alternate constructor for `meteredTurtle` as follows.

```
  meteredTurtle(double x, double y) : Turtle(x,y) {
    distance = 0;
  }
```

With this constructor, the metered turtle would be created at position `(x,y)` on the screen.

We now discuss destructors, in the general setting. As before suppose we have a class `B` which inherits from class `A`. Then the destructor for class `B` should be used to destroy the new data members introduced in `B` that were not present in `A`. The data members inherited from `A`? These would be destroyed by an implicit call that would get made to the destructor of `A` at the end of the execution of the call to the destructor of `B`. You should not explicitly make a call to the destructor of `A` from inside the destructor of `B`!

The general rule is: destruction happens automatically, in reverse order of creation. In the exercises you will experiment with code which will illustrate these ideas.

19.3.3 Polymorphism and virtual functions

A piece of code is said to be *polymorphic* if it can be executed for variables of more than one type, or more than one class. Clearly, once we use inheritance, the code of the superclass becomes polymorphic because it can be executed for objects of the superclass and also the subclass. This raises some subtle questions, which we explore next.

Consider the following code.

```
class Animal{
public:
  void whoAmI(){ cout << name() << endl; }
  string name(){ return "Animal"; }
};

class Mammal: public Animal{
```

```
public:
  string name(){ return "Mammal"; }
};

int main(){
  Animal a;
  Mammal b;
  a.whoAmI();
  b.whoAmI();
}
```

Executing `a.whoAmI()` will clearly cause "Animal" to be printed out. More interesting is the execution of `b.whoAmI()`. What should it print? The call `b.whoAmI()` is to the inherited member function `whoAmI` in the superclass `Animal`. That function `whoAmI` calls `name`, but the question is which `name`. Will it be the `name` in `Animal` or in `Mammal`?

The answer turns out to be the `name` in `Animal`, and thus in this case, "Animal" will be printed. There is a natural reason for this. When class `Animal` is defined, the reference to `name` in the body of `A::whoAmI` is understood by C++ to mean the `name` in `Animal` itself. This understanding is not changed when `Mammal` is defined.

But you might suppose that it should be useful if the `name` in `Mammal` gets used instead. C++ provides a way to accomplish it: add a single keyword `virtual` to the definition of `name` in `Animal`. Thus the class would be defined as:

```
class Animal{
public:
  void whoAmI(){ cout << name() << endl; }
  virtual string name(){   // note the added keyword virtual
    return "Animal";
  }
};
```

This keyword says that the definition of `name` should not be treated as a unique, final definition. It is possible that `name` might be over-ridden in a subclass, and if so, that definition of `name` which is most appropriate (most derived!) for the object on which the call is made should be considered. When we call `b.whoAmI()`, the most appropriate definition for `name` is the one in `Mammal`, since `b` is of type `Mammal`. So that definition gets used, and our code will now indeed print "Mammal". Note that `a.whoAmI()` will continue to print "Animal".

19.3.4 Polymorphism and pointers

C++ also allows polymorphic pointers. This feature, which we explain next, is an extremely important part of inheritance. We will see its utility immediately in Section 19.4

Suppose that `aptr` is a variable of type `A*`, where `A` is a class. Suppose `B` is a subclass of `A`. Then we can store addresses of instances of `A` as well as of `B` (or even of instances of subclasses of `B` and so on) in `aptr`. In other words, the following code is legal.

```
class A{ ... };
class B : public A{...};
B b;
A* aptr;
aptr = &b;
```

The pointer variable aptr is said to be polymorphic because it can contain pointers to objects of more than one class.

Suppose now that f is a member function in A which has been overridden in B. Suppose that f does not have any parameters. Then suppose we execute:

```
aptr->f();
```

Which version of f will get executed? The situation is analogous to the previous section. If we declared f to be virtual, then the code for f defined in B will get executed. Otherwise the code from A will get executed.

19.4 Program to print past tense

Suppose you wish to write a program that takes as input a verb from the English language, and prints out its past tense. Thus, given "play", the program must print "played", given "write", the program must print "wrote", and so on. A simple implementation would be to store every verb and its past tense as strings in memory. Given the verb, we can then print out the corresponding past tense.

But you will perhaps observe that for most verbs, the past tense is obtained simply by adding a suffix "ed", as is the case for the verbs "play", "walk", "look". We may consider these verbs to be *regular*. Verbs such as "be", "speak", "eat" which do not follow this rule could be considered *irregular*. Thus it makes sense to store the past tense form explicitly only for irregular verbs; for regular verbs we could simply attach "ed" when asked. This can be programmed quite nicely using inheritance.

We define a class verb that represents all verbs; it consists of subclasses regular and irregular respectively. The definition of verb contains information which is common to all verbs. The definition of regular adds in the extra information needed for regular verbs, and similarly the definition of irregular.

```
class verb{
protected:
  string root;
public:
  string getRoot(){return root;}
  virtual string past_tense(){return ""};
};
```

The member root will store the verb itself. We have defined the member function past_tense to be *virtual*. For now it returns the empty string. But this is not important, since we expect to override it in the subclasses.

The subclasses regular and irregular are as you might expect.

```
class regular : public verb{
public:
  regular(string rt){
    root = rt;
  }
  string past_tense(){return root + "ed";}
};

class irregular : public verb{
  string pt; // past tense of the verb
public:
  irregular(string rt, string p){
    root = rt;
    pt = p;
  }
  string past_tense(){return pt;}
};
```

Thus, to create an instance `v1` that represents the verb "play" we would just write

```
regular v1("play");
```

After this if we wrote `v1.past_tense()`, we would get the string `"played"` as the result. Similarly

```
irregular v2("be","was");
```

would create an instance `v2` to represent the verb "be". And of course `v2.past_tense()` would return "was".

We now see how the above definitions can be used to write a main program that returns the past tense of verbs. Clearly, we will need to somehow store the information about verbs. For this, we use a vector. We cannot have a single vector storing both regular and irregular verbs. However, we can define a vector of pointers to `verb` in which we can store pointers to `irregular` as well as `regular` objects. Thus the program is as follows.

```
int main(){
  vector<verb*> V;
  V.push_back(new regular("watch"));
  V.push_back(new regular("play"));
  V.push_back(new irregular("go","went"));
  V.push_back(new irregular("be","was"));

  string query;
  while(cin >> query){
    size_t i;
    for(i=0; i<V.size(); i++)
      if (V[i]->getRoot() == query){
        cout << V[i]->past_tense() << endl;
        break;
      }
    if(i == V.size()) cout << "Not found.\n";
  }
}
```

We begin by creating the vector `V`. We then create instances of regular and irregular verbs on the heap, and store pointers to them in consecutive elements of `V`. Finally, we enter a loop in which we read in a `query` from the user, check if it is present in our vector `V`. If so, we print its past tense. Note that if the `for` loop ends with `i` taking the value `V.size()`, it must be the case that no entry in `V` had its `root` equal to the `query`. In this case we print `"Not found."`. As you know, the while loop will terminate when `cin` ends, e.g. if the user types control-d.

A number of points are to be noted regarding the use of inheritance in this example.

1. Our need was to represent the category of verbs which consisted of mutually disjoint sub-categories of irregular and regular verbs. This is a very standard situation in which inheritance can be used.

2. The vector `V` is polymorphic: it can contain pointers to objects of type `irregular` as well as of type `regular`. We can invoke the operation `past_tense` on objects pointed to by elements of `V`, without worrying about whether the objects are of type `regular` or `irregular`. Because `past_tense` is virtual, the correct code gets executed.

19.5 Abstract Classes

You will note that we return the empty string in the `past_tense` function in `verb`. Returning an empty string does not make sense, but we did this because we expected that the `verb` class would never be used directly; only its subclasses would be used in which the function would get overridden. This idea works, but it is not aesthetically pleasing that we should need to supply an implementation of `past_tense` in `verb` expecting fully well that it will not get used.

One possibility is to only declare the member function `past_tense` in `verb`, and not supply any implementation at all. Unfortunately, whenever an implementation is not supplied, the compiler expects to find it somewhere, in some other file perhaps. In other words, the compiler expects that somewhere else we would supply the implementation

```
string verb::past_tense(){
  // implementation
}
```

If such a definition is not given the compiler or the linker will produce an error message.

What we need is a way to tell the compiler that we do not at all intend to supply an implementation of `past_tense` for the `verb` class. This is done by suffixing the phrase “`= 0`” following the declaration. Thus we would write

```
class verb{
 ...
 public:
  virtual string past_tense() = 0;
 ...
}
```

Writing "`= 0`" following the declaration of a member function tells the compiler that we do not intend to at all supply an implementation for the function. You may think of 0 as representing the `NULL` pointer, and hence effectively indicating that there is no implementation.

There is an important consequence to assigning a function to 0. Suppose a class `A` contains a member function `f` assigned to 0. Then we cannot create an instance of `A`! This is because for that instance we would not know how to apply the function `f`. In C++, classes which cannot be instantiated are said to be *abstract*. Indeed, the only way of making a class abstract is to assign one of its member functions to 0.[2]

So in our case, if we assign `past_tense` to 0 in `verb`, then the class `verb` would become abstract. We would not be able to create instances of it. But this is fine, we indeed would not like users to instantiate `verb`, instead we expect them to instantiate either `regular` or `irregular`.

19.6 Multiple inheritance

Sometimes, an object can be thought of as a specialization of not one, but *two* other objects. For example, we might have an `Automobile` class and a `SolarPoweredDevice` class.

```
class Automobile{
  double mileage;
};
class SolarPoweredDevice{
  double cellEfficiency;
};
```

Suppose we also need to represent solar powered automobiles, they would need to have features of both the `Automobile` class as well as `SolarPoweredDevice` class. We can get this by constructing a class `SolarPoweredAutomobile` which inherits from both!

```
class SolarPoweredAutomobile : public Automobile,
                               public SolarPoweredDevice {};
```

Now instances of the `SolarPoweredAutomibile` class would have members `mileage` as well as `cellEfficiency`, as you might expect. Function members would also be inherited from all the superclasses, as many as there might be.

We will see some substantial examples of multiple inheritance in Chapter 22.

[2] If we make the constructor of a class `private` and there are no member functions that use the constructor, then effectively we have ensured that the class cannot be instantiated. But technically such classes are not considered to be abstract.

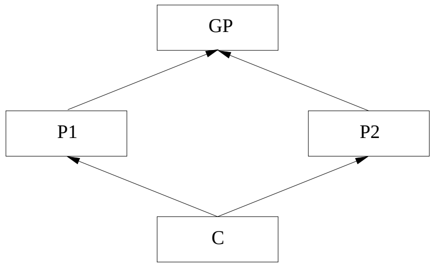


Figure 19.1: Diamond inheritance. Arrows go from child to parent.

There are some obvious problems: what happens if the parent classes P1, P2 of a class C both have a member with the same name m? In this case, the child class would get two copies of the member, and you would have to refer to the copies as P1::m and P2::m.

19.6.1 Diamond Inheritance

It is also possible that P1, P2 inherit from a common base class GP. In which case, the class C will actually get two copies of the inherited object GP, corresponding to P1 and P2.

This case, in which we derive a class C by inheriting from parent classes P1, P2, which in turn inherit from a single class GP is said to constitute *Diamond inheritance*. This is because the pictorial representation of the inheritance has the diamond shape, Figure 19.1.

However, sometimes when we have diamond inheritance, we might want to have only one copy of the inherited object instead of one from each parent. It is possible to do this in C++ by specifying the derivation of P1, P2 from GP as virtual. Thus we would write:

```
class GP{ double x; };
class P1: virtual public GP{};
class P2: virtual pubilc GP{};
class C: public P1, public P2{};
```

With this, the class C will contain only one copy of the inherited object GP. Note that this inherited object will be initialized directly by calling its constructor. Thus if the initialization list of P1 or P2 contains a call to the constructor of GP, then those calls will be ignored.

19.7 Exercises

1. Implement the meteredTurtle class without using inheritance.

2. Suppose you have a class V defined as

```
class V{
  double x,y,z;
public
  V3(double p, double q, double r){ x=p; y=q; z=r; }
}
```

Define a class W that inherits from V and has a member function dot which computes the dot product of two vectors. Thus given V type objects v,w, then the dot product is v.x * w.x + v.y * w.y + v.z * w.z. Be careful that you only use the constructor provided in V.

3. Define a class realTurtle such that realTurtle objects move with some specifiable speed when they move. They should also turn slowly.

4. What do you think will happen when you execute the program given below? Run it and check if you are right.

```
class A{
public:
  A(){cout << "Constructor(A).\n";}
  ~A(){cout << "Destructor(A).\n";}
};

class B: public A{
public:
  B(){cout << "Constructor(B).\n";}
  ~B(){cout << "Destructor(B).\n";}
};

class C: public B{
public:
  C(){cout << "Constructor(C).\n";}
  ~C(){cout << "Destructor(C).\n";}
};

int main(){
  C c;
}
```

5. What will the following code print?

```
struct A{
  virtual int f(){return 1;}
  int g(){return 2;}
};

struct B : public A{
  int f(){return 3;}
  int g(){return 4;}
}

A* aptr;
```

```
aptr = new B;
cout << aptr->f() <<' '<< aptr->g() << endl;
```

int f() return 3;

int g()return 4;;

A *aptr;

aptr = new B;

cout ¡¡ aptr-

6. Write the past tense generation program using just two classes, a `verb` class and an `irregular` class. A regular verb should be created as an instance of `verb`, and an irregular as an instance of `irregular`.

Chapter 20

Inheritance based design

Inheritance is often very useful in designing large complex programs. Its first advantage we have already discussed: inheritance is convenient in representing categories and subcategories. But there are some related advantages which have to do with the management of the program development process. We will discuss these next, and then in the rest of the chapter we will see some examples.

There are many approaches to designing a large program. A classical approach requires that we first make a complete study of the program requirements, and only thereafter start writing the program. More modern approaches instead acknowledge/allow for the possibility that requirements may not be understood unless one has built a few versions of the program. Also, if a program works beautifully, users will inevitably ask that it be enhanced with more features. In any case, programs will have a long lifetime in which the requirements will evolve. So the modern approaches stress the need to design programs such that it is easy to change them. As we have discussed, the whole point of inheritance is to build new classes out of old, and this idea will surely come in useful when requirements change.

Even if the requirements are well understood and fixed (at least for that time in the program development process), designing a large program is tricky. It greatly helps if the program can be designed as a collection of mostly independent functions or classes which interact with each other in a limited, clearly defined manner. Such partitioning is helpful in understanding the behaviour of the program and also checking that it is correct: we can consider testing the parts separately for example. But it also has another advantage: different programmers can work on the different parts simultaneously. As we will see, inheritance based designs have much to offer in this regard also.

Another modern programming trend is the use of *components*. Most likely, to write professional programs you will not use bare C++ (or any other programming language), but will start from a collection of functions and classes which have been already built by others (for use in other, similar projects). You will adapt the classes for your use, and as we have seen, this adaptation is natural with inheritance.

In the rest of this chapter we will see two case studies. First we revisit our formula drawing program. We will rewrite it using inheritance. It will turn out that this way of writing makes it easier to maintain and extend. Next we will discuss the design of the graphics in `simplecpp`. Inheritance plays a substantial role in its design. Finally, we will see the `composite` class, which will enable you to define new graphical objects which are made by composing the simple graphics objects we know so far.

20.1 Formula drawing revisited

Consider the formula drawing problem from Section 17.1: given a mathematical formula, draw it on the graphics canvas in a nice manner. In Section 17.1 our specific goal was to draw the formula such that operands in sums, differences and products were drawn left to right horizontally, while the dividend and the divisor were drawn above and below a horizontal bar respectively. In this section we will see how the program can written in using inheritance. A benefit of this will be that it will be easy to extend the program to include new operators. To illustrate this we will implement the exponentiation operator which requires the exponent to be written above and to the right of the base.

For simplicity, we ignore the problem of accepting input from the keyboard: we will assume that the formula to be drawn is given as a part of the program, i.e. to draw $\frac{a}{b+c}$ we will first construct it in our program by writing something like:

```
Node f2('/', new Node("a"),
             new Node('+', new Node("b"), new Node("c"))
           );
```

and then we can call `f2.setSize()` and so on.

20.1.1 Basic design

The use of inheritance is natural if the entities we want to represent form a category and subcategories thereof. In the present case, we wish to represent mathematical formulae. These formulae can further be classified based on the top level operators: for example in the formula $\frac{a}{b+c}$, the last operation to be performed is division, and hence we will designate it to be in the class `Div`. In the formula $a + \frac{b}{c}$, the last operation to be performed is addition, so we will designate it to be of type `Sum`. So we have the general category of mathematical formulae, and subcategories `Sum, Diff` (difference), `Prod` (product), and `Div`, based on the last operation performed to evaluate the formula. Naturally, we will have a class corresponding to general category, from which the classes for the subcategories will be inherited.

We will use the class `Formula` to represent all mathematical formulae. This will be analogous to the class `Node` of Section 17.1.2. This class will have a subclass `Formula2` which will represent all mathematical formulae in which the top level operator is binary. This class will have subclasses `Sum`, `Diff`, `Prod` and `Div` respectively representing formulae in which the top level operators are +, -, ×, and ÷. Remember, that in the Exercises of Chapter 17 we pointed out that it helps to consider unary and ternary formulae – you can think of the class `Formula2` as strictly contained in `Formula`.

The algorithm for drawing the formula will be the same; hence we will have methods `setSize` and `draw` in all classes. These will be defined differently depending upon the type of the operator.

Here is the definition of the class `Formula`.

```
class Formula{
protected:
  double width, height, descent;
public:
  virtual void setSize()=0;
  virtual void draw(float clx, float cly)=0;
  double getWidth(){return width;}
  double getHeight(){return height;}
  double getDescent(){return descent;}
};
```

We do not expect `Formula` to be instantiated, so we have declared some of its methods to be pure virtual. Also note that in Section 17.1.2, we used structures instead of classes. Thus everything was public. Now we are more careful about making members visible and hence we have defined accessor functions for `width, height` and `descent`.

We next define the class of formulae with 2 operands at the top level.

```
class Formula2 : public Formula{
protected:
  Formula* lhs;
  Formula* rhs;
  virtual string op()=0;
};
```

Instead of storing the operator explicitly, we are using a function `op` which will return it. The function will be defined only in classes in which the operator is known.

Next we define the subclass `Formula2h` of expressions which require a horizontal layout.

```
class Formula2h : public Formula2{
public:
  void setSize(){
    lhs->setSize();
    rhs->setSize();
    descent = max(lhs->getDescent(), rhs->getDescent());
    width = lhs->getWidth() + textWidth(op()) + rhs->getWidth();
    height = descent + max(lhs->getHeight() - lhs->getDescent(),
                           rhs->getHeight() - rhs->getDescent());
  }
  void draw(float clx, float cly){
    lhs->draw(clx,cly);
    rhs->draw(clx+lhs->getWidth()+textWidth(op()), cly);
    Text(clx+lhs->getWidth()+textWidth(op())/2,cly, op()).imprint();
  }
};
```

These function implementations are similar to the case `'+'` of the corresponding functions on `Node`(Section 17.1.4). The only difference is that we are using accessor functions instead of the members `height, width, descent` and `op` is not a data member but a function member.

We can now create classes to represent sums, differences, and products.

```
class Sum :  public Formula2h{
public:
  Sum(Formula* lhs1, Formula* rhs1){ lhs = lhs1; rhs = rhs1; }
  string op(){ return "+";}
};

class Diff :  public Formula2h{
public:
  Diff(Formula* lhs1, Formula* rhs1){ lhs = lhs1; rhs = rhs1; }
  string op(){ return "-";}
};

class Prod : public Formula2h{
public:
  Prod(Formula* lhs1, Formula* rhs1){ lhs = lhs1; rhs = rhs1; }
  string op(){ return "*";}
};
```

These classes merely give a constructor, and the operator. Notice that the layout aspects have been dealt with in the class `Formula2h`.

We could analogously define an `Formula2v` class, which does vertical layout of its operands. However, it is unlikely there will be many operators requiring a vertical layout with a separating horizontal bar. So we directly define the `Div` class.

```
class Div : public Formula2{
  static const float Bheight = 20;  //space for horizontal bar.
public:
  Div(Formula* lhs1, Formula* rhs1){ lhs = lhs1; rhs = rhs1; }
  string op(){return "/";}
  void draw(float clx, float cly){
    Line(clx,cly,clx+width,cly).imprint();
    lhs->draw(clx+width/2-lhs->getWidth()/2,cly-Bheight/2-lhs->getDescent());
    rhs->draw(clx+width/2-rhs->getWidth()/2,cly+
      Bheight/2+rhs->getHeight()-rhs->getDescent());
  }
  void setSize(){
    lhs->setSize(); rhs->setSize();
    width = max(lhs->getWidth(), rhs->getWidth());
    height = lhs->getHeight() + Bheight + rhs->getHeight();
    descent = rhs->getHeight() + Bheight/2;
  }
};
```

This corresponds to the code for the case `'/'` in the corresponding functions on `Node` (Section 17.1.4). It also defines a constructor and the function `op`.

Finally, we need a class to represent literals, i.e. numbers or variables given in the formula.

```
class Literal : public Formula{
  string value;
public:
  Literal(string v){value=v;}
  void setSize(){
    width = textWidth(value);
    height = textHeight(); descent = height/2;
  }
  void draw(float clx, float cly){
    Text(clx+width/2,cly,value).imprint();
  }
};
```

This corresponds to the code for the case `'P'` in the corresponding functions on `Node` (Section 17.1.4).

We can now give a simple main program which can use the above definitions to render the formula $1 + \frac{2}{\frac{451}{5}+35}$.

```
int main(){
  initCanvas("Formula drawing");

  Sum e(new Literal("1"),
        new Div(new Literal("2"),
                new Sum(new Div(new Literal("451"),new Literal("5")),
                        new Literal("35"))));

  e.setSize();
  e.draw(200,200+e.getHeight()-e.getDescent());

  getClick();
}
```

20.1.2 Comparison of the two approaches

At first glance, it might seem that the inheritance based approach is more verbose than the approach of Section 17.1. This is true, but the verbosity has bought us many things.

A key improvement is that we have partitioned the program into manageable pieces. The code of Section 17.1 had just one class. All the complexity was placed into that class. In contrast, in the new code, different concerns are separated into different classes. For example, the class `Formula2` only models the fact that formulae can have two operands, nothing more. The class `Formula2h` shows how to layout formulae requiring horizontal layout. We can place each class into its header and implementation files, and the main program into a separate file, if we wish. This way, if we wish to change something regarding a certain issue (e.g. horizontal layout) we know that we will likely modify only one small file. This is a big benefit of the new approach.

Another important benefit arises when we consider adding new functionality to the program. Suppose we want to implement layouts of exponential expressions. As you will see, we can do this without touching any of our old files (except the main program file, if we wish to use exponential expressions, of course). The key benefit of this strategy is: we can be sure that when we add exponential expressions, *there isnt even a remote chance of damaging the old working code.* Programmers (deservedly) tend to be paranoid about their code, and this kind of reassurance is useful. Notice that if the old code was written by one programmer, and the new one by another, then it is very convenient if one programmer's code is not touched by another. This way there is clarity about who was responsible for what.

20.1.3 Adding exponential expressions

We will add a class `Pow` that will represent exponential expressions. This will be a subclass of `Formula2` since it has 2 operands.

```
class Pow : public Formula2{
public:
  Pow(Formula* lhs1, Formula* rhs1){ lhs = lhs1; rhs = rhs1; }
  string op(){return "^";}
  void draw(float clx, float cly){
    lhs->draw(clx,cly);
    rhs->draw(clx+lhs->getWidth(),
              cly - (lhs->getHeight() - lhs->getDescent())
                  - rhs->getDescent()
       );
  }
  void setSize(){
    lhs->setSize(); rhs->setSize();
    width = lhs->getWidth() + rhs->getWidth();
    height = lhs->getHeight() + rhs->getHeight();
    descent = lhs->getDescent();
  }
};
```

The basic idea is to layout the exponent above and to the right of the base. The detailed expressions which decide how to position what are obtained in the manner of Section 17.1.4, and are left for you to figure out. Using this class is simple, to represent $X + Y^Z$ we simply write `Sum(new Literal("X"), new Pow(new Literal("Y"), new Literal("Z")))`.

The key point to appreciate is that this new code can be developed independently, in a new file, without even having the rest of the code, only the class header files would be needed.. If we had used the coding approach of Section 17.1, we would need to have and modify the old code.

20.2 The simplecpp graphics system

We will discuss the role played by inheritance in the design of the simplecpp graphics system. Although this system is quite small by standards of real graphics systems, we will not discuss the entire system here, but only some relevant portions of it.

Briefly stated, the core specification of the system is: allow the user to create and manipulate graphical objects on the screen. This statement is very vague, of course. What does it mean to *manipulate* objects? As you know, in simplecpp, manipulate simply means move, rotate, scale. There are also other questions: what kind of primitives is the user to be given? Will the user need to specify how each object appears in each *frame* (like the picture frames in a movie) or will the user only state the incremental changes, e.g. move object x, which means the other objects remain unchanged? As you know, we have opted for the latter. And then of course there is the question of what kinds of objects we can have. As you know, simplecpp supports the following kinds of graphical objects: circles, lines, rectangles, polygons, turtles and text.

So in the rest of this section, we consider the problem of creating and manipulating the objects given above, in the manner described above. We will not discuss issues such as the pens associated with each object, or graphical input, as in the getClick command.

Clearly, such a system must have the following capabilities:

1. It must be able to keep track of the objects created by the user so far.

2. It should be able to display the objects on the screen.

3. It should be able to manipulate the objects as requested, e.g. move an object.

Let us take item 2 first. simplecpp is built on top of the X Windows system, which provides functions that can draw lines, arcs, polygons, circles (filled and non-filled) on the screen, at the required place. So we call these functions. Item 3 is also relatively easy. With each object we associated some configuration data, which says how large it is to be drawn, in what orientation, and where. Note that this data is distinct from the shape data. Item 1 is also not difficult in principle: we merely keep a list or a set of some kind in which we place each object. To complete this very high level description we need to answer one more question: when should the objects be displayed? The simplest answer to this is: whenever the user changes the state of any object, clear the entire display and display all objects again. Inheritance is useful for facilitating many of the actions described here.

It should be clear that we have the category of all graphical objects, and then subcategories corresponding to different types of objects, e.g. circles. So clearly, these correspond to subclasses of the class representing the category ALL of all objects. Our categories indeed form a heirarchy, and so do the associated classes, as shown in Figure 20.1. The class associated with the category of all objects is called Sprite, in honour of the Scratch programming environment (scratch.mit.edu), where the name is used for a similar concept.

The heirarchy facilitates storing of information as follows. In the Sprite class we keep all attributes that are common to all graphics objects. At first glance, you might think that precious little might be common to all the very different looking objects: circles, lines, polygons and so on. But as mentioned earlier, when a graphics object is to be displayed on the screen, it will have a position, orientation, scale in addition to its shape. It will also have

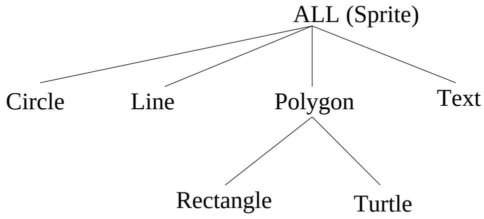


Figure 20.1: Heirarchy of graphics object categories

attributes such as colour. All objects will have these attributes. So these attributes become data members in the `Sprite` class.

The classes `Circle, Line, Polygon` etc. will contain the shape related attributes (in addition to all the attributes inherited from `Sprite`). For example, `Circle` contains a data member called `radius` which holds the radius of the circle being drawn. The `Polygon` class contains an array which holds the coordinates of the vertices of the polygon. As discussed earlier these coordinates are given in a specially created coordinate frame; while drawing they will be drawn relative to the position of the polygon (as recorded in the `Sprite` part of its object). Figure 20.2 shows possible ways contents of the `Circle` and `Sprite` classes. The actual implementations different, and you can see them in the code supplied.

In addition, we must also consider function members. Suppose we wish to move an object. This requires two actions: (a) recording that the object has indeed been moved, and updating its position accordingly, (b) redrawing the object on the screen. Clearly, action (a) can be performed independent of the shape of the object, whereas (b) requires the shape information. Thus in our implementation, action (b) is implemented by a `paint` method in each shape class. The `move` member function is defined in the `Sprite` class. It performs action (a) using the attributes available in the `Sprite` class. It then signals that all objects need to be redrawn.

The redrawing works as follows. Essentially `simplecpp` maintains a vector that holds pointers to all objects active at the current instant. Suppose the vector is named `ActiveSprites`, then its declaration would be

```
vector<Sprite*> ActiveSprites;
```

Because the elements of `ActiveSprites` have type `Sprite*`, they can hold pointers to any graphical object. When the objects are to be drawn, we simply iterate over the queue and execute the `paint` method of the object.

```
  for(int i=0; i < ActiveSprites.size(); i++){
    ActiveSprites[i]->paint();
  }
```

The `paint` method is virtual, and so the `paint` method in the class of the object is used. This is similar to the way we used a vector of `Verb*` to store `regular` and `irregular` objects and invoked the `past_tense` virtual function on them in Section 19.4.

In summary, inheritance gives us three main benefits. It is easy to organize our data strorage manner: the `Sprite` class stores the configuration related data and the other classes

```
class Sprite{
protected:
  double x,y;          // position on the canvas
  double orientation;  // angle in radians made with the x axis
  double scale;        // scaling factor
  Color fill_color;    // Color is a data type in the X windows Xlib package.
  bool fill;           // whether to fill or not
  ...
 public:
  Sprite();
  Sprite(double x, double y);
  ...
  void forward(double dist);
  ...
  virtual void paint()=0;
  ...
}

class Circle : public Sprite{
 private:
  double radius;
 public:
  Circle();
  Circle(double x, double y, double radius=10);
  void init(double x, double y, double radius=10);
  virtual void paint()
};
```

Figure 20.2: Possible definitions of `Sprite` and `Circle`

store the shape related data. Also because of polymorphism and virtual functions, we can store pointers to different types of graphical objects (but only subtypes of `Sprite`) in a single vector, and iterate over the vector. Finally, we can add new shapes easily: we simply define a new shape class which is a subclass of `Sprite`, without having to modify any existing code.

We have somewhat simplified the description of `simplecpp` graphics in order to explain the use of inheritance. The actual system is more sophisticated. We see an example of this sophistication next.

20.3 Composite graphics objects

We discuss how you can define a class whose instances are composite, i.e. a single instance can contain several simple objects. Once built, you can use the class in your program to create instances. In defining a composite object you need inheritance as we will see.

Suppose you want to draw many cars on the screen. It would be nice if you could design a class `Car` which you could then instantiate to make many cars. A car is a complex object: it cannot be drawn nicely using just a single polygon, or a single circle. It will require several simple objects that `simplecpp` provides. These simple objects will have to be grouped together, and often be manipulated together, e.g. if we want the car to move, we really mean to move all its constituent parts. The class `Composite` which we discuss next, will allow you to group together objects. We can then define a `Car` class by inheriting from the `Composite` class.

Our `Composite` class primarily serves as a ”container” to hold other graphics objects. It has a frame of reference, relative to which the contained objects are specified. The `Composite` class has been defined as a subclass of the `Sprite` class. Thus it inherits member functions such as move, forward, rotate from the Sprite class. The `Composite` class is designed so that the member functions will cause the contained objects to respond appropriately, i.e. when you move a `Composite`, everything inside gets moved. However, you can override these methods if you wish. For example, suppose your composite object consists of the body of a car and its wheels. When you call `forward` on this, by default everything will go forward. You might want the wheels to rotate in addition to moving forward. This can be accomplished by overridding. You can also additionally define your own new member functions which do new things. For example, the car might have a light on the top and there could be a member function which causes the light to change colour from white to yellow (suggesting it is switched on). This could be done using a new member function.

Using the `Composite` class is fairly easy. There are only three important ideas to be understood: the notion of ownership, and the `Composite` class constructor.

20.3.1 Ownership

A detail we have hidden from you so far is: every graphics object has an “owner”. When we say that object `X` owns object `Y`, we merely mean that object `Y` is specified relative to the coordinate frame of object `X`. For the objects you have been creating so far, the owner was the canvas: the objects were drawn in the coordinate frame of the canvas. When an object is created as a part of a composite object, it must be drawn relative to the frame of the composite object, and hence must be owned by the composite object. Thus an important step in defining a composite object is to declare it to be the owner of the contained objects.

To do this, the constructor of every graphical object is provided with an optional argument named `owner`. This argument takes value `NULL` by default which `simplecpp` interprets to mean the canvas. Thus so far we did not tell you about this argument, and you didnt not specify the value, and hence `simplecpp` made the canvas the owner of all the objects you created. If you want to indicate a different owner, you instead pass a pointer to that owner. Since we create contained objects in the body of the composite, we must pass a pointer to the composite object itself. As you know, inside the definition of an object, the keyword `this` denotes a pointer to the object. So the extra argument must have `this` as its value.

20.3.2 The `Composite` class constructor

The `Composite` class constructor has the following signature.

```
  Composite(double x, double y, Composite* owner=NULL)
```

Here the last argument `owner` gives the owner of the composite object being defined, and `x,y` give the coordinates of the composite object in the frame of its owner. As mentioned earlier, if you do not specify this argument, it is taken as NULL, indicating that the canvas is the owner. The `owner` argument must be specified if this composite object is itself a part of another composite object. This kind of containment is allowed and we will see an example shortly.

20.3.3 A `Car` class

We now construct a `Car` class. Our car will consist of a polygonal body, and two wheels. We will give the wheels some personality by adding in spokes. So we will model a car as a composite object, consisting of the body and the wheels. But note that a wheel itself contains a circle representing the rim, and lines representing the spokes. So the wheel will itself have to be represented as a composite object. Note that we allow one composite object (e.g. a car) to contain other ordinary objects (e.g. body) or other composite objects (e.g. wheels).

We begin by defining a class for wheels.

```
class Wheel : public Composite{
  Circle *rim;
  Line *spoke[10];
public:
  Wheel(double x, double y, Composite* owner=NULL) : Composite(x,y,owner) {
    rim = new Circle(0,0,50,this);
    for(int i=0; i<10; i++){
      spoke[i] = new Line(0, 0, 50*cos(i*PI/5), 50*sin(i*PI/5), this);
    }
  }
};
```

There are two private data members. The member rim which is defined as a pointer to the Circle object which represents the rim of the wheel. Likewise spoke is an array of pointers to each spoke of the wheel. The objects themselves are created in the constructor. This is a very common idiom for defining composite graphics objects.

The constructor customarily takes as argument a pointer to the owner of the composite object itself, and the position of the composite object in the frame of the owner. It is customary to assign a default value NULL for the owner parameter, as discussed earlier. The initialization list Composite(x,y,owner) merely forwards these arguments so that the core composite object is created at the required coordinate and gets the specified owner. Inside the constructor, we create the sub-objects. So we create the circle representing the rim, and as you can see we have given it an extra argument this, so that the Wheel object becomes the owner of the rim. Likewise we create lines at different inclinations to represent the spokes, and even here the extra argument this causes the lines to be owned by the Wheel object.

The Car class can be put together by using Wheel instances as parts.

```
class Car : public Composite{
  Polygon* body;
  Wheel* w1;
  Wheel* w2;
public:
  Car(double x, double y, Color c, Composite* owner=NULL)
    : Composite(x,y,owner){
    double bodyV[9][2]={{-150,0}, {-150,-100}, {-100,-100}, {-75,-200},
                        {50,-200}, {100,-100}, {150,-100}, {150,0}, {-150,0}};
    body = new Polygon(0,0, bodyV, 9, this);
    body->setColor(c);
    body->setFill();
    w1 = new Wheel(-90,0,this);
    w2 = new Wheel(90,0,this);
  }
  void forward(double dx, bool repaintP=true){
    Composite::forward(dx,false);  // superclass forward function
    w1->rotate(dx/50,false); // angle = dx/radius
    w2->rotate(dx/50,false);
    if(repaintP) repaint();
  }
};
```

As will be seen, the private members are the pointers to the body and the two wheels. In the constructor, the body is created as a polygon. We have provided a parameter in the constructor which can be used to give a colour to the body. Finally, the wheels are created. For all 3 parts, the last argument is set to this, because of which the parts become owned by the Car object, as we want them to be.

The definition also shows the forward function being overridden. As discussed, we want the car to move forward, which is accomplished by calling the forward function of the superclass. But we also want the wheels to turn; this is accomplished by rotating them. Clearly, if the car moves forward by an amount `dx`, then the wheels must rotate by `dx/r` radians, where `r` is the radius of the wheels. But why does the `rotate` function called with an extra argument `false`? And why is the function `repaint` called? We explain these next.

20.3.4 Frames

Another detail of `simplecpp` graphics which we have withheld from you is that all the configuration change commands, i.e. `forward`, `move`, `rotate` and so on have an additional argument `repaintP` which takes the default value `true`. If `repaintP` is `true`, then the canvas is repainted as discussed in Section 20.2 after every configuration change. If `repaintP` is `false`, then the repainting is not done, only the configuration change is recorded.

This feature is useful especially when invoking a configuration change command on subobjects comprising a composite object. We do not want repainting to happen after a move of each subobject. It is inefficient and also causes visually annoying. Rather we want repainting to happen once, after the configuration change is recorded for all subobjects. This is what the code above accomplishes: no repainting happens after the `Composite::forward` as well as the two `w...->rotate` operations. Repainting is done only at the end unless it is disabled by the caller to `Car::forward`.

20.3.5 Main program

Finally, here is a main program that might use the above definitions.

```
int main(){
  initCanvas("Car",0,0,800,800);

  Car c(300,300,COLOR("blue"));
  Car d(300,600,COLOR("red"));
  d.scale(0.5);
  getClick();

  for(int i=0; i<100; i++){
    c.forward(3.0,false);
    d.forward(1.5,false);
    repaint();
  }
  getClick();
}
```

The main program creates two cars, one blue and another red. The red car is then scaled to half its size. This causes all the components of the car to shrink – this is handled automatically by the code in the `Composite` class.

Finally, we move the cars forward. When you execute this, the car wheels should appear to roll on the ground. Further, the wheels of the smaller car should appear to have twice as many rotations per unit time because the smaller wheel has half the radius but is travelling the same distance as the larger wheel.

Exercises

1. To the program of Section 20.1 add the capability of drawing summation formulae, i.e. formulae
$$\sum_A^B C$$
where A, B, C can themselves be formulae.

2. Define a composite object to model a face. Define methods for showing emotions, e.g. smiling. Use your imagination.

3. An attractive idea in the Scratch programming system is of *costumes* for sprites. A costume is merely a description of how a sprite can appear. For example, a bird sprite might have two costumes: one in which the wings are together, and another in which the wings are spread out. By alternating between the two costumes as the bird moves, you can create the effect of the bird flying. Design this sprite.

4. Do you think it will be possible to design a class `CostumedObject` which can make it easier to define multiple costumes? If so do it.

5. You are to write a program using which non-programmers can write simple animations. The input to the program could be something like the following sequence of commands.

```
Circle 100 200 5
Rectangle 200 300 40 50
Circle 100 200 15
Move 0 50 50
Left 1 30
Move 2 70 -70
Wait 0.5
...
```

In this, the first 3 lines defined 3 objects. As you might guess, the numbers following the shape names are the arguments for the corresponding constructors. For the rest of the sequence, the constructed objects respectively get numbered 0, 1, 2 (or till as many objects as we have defined). In the rest of the command sequence, a graphical object will be referred to by its number. Thus the command *Move 0 50 50* causes object 0 (the first circle) to be displaced by 50, 50 along x and y directions. Likewise `Left 1 30` causes the rectangle to be rotated left by 30 degrees. You may also define commands to wait for specified amount of time.

Write the program. Note that `Sprite` objects cannot be stored in a `vector`. However, you can create the `Sprite` on the heap and store a pointer to it in a vector.

Chapter 21

Discrete event simulation

We have already discussed the general notion of simulation: given the current state and laws of evolution of a system, predict the state of the system at some later date. In Chapter 15, we considered the simulation of heavenly bodies, as might be required in astronomy. However, simulation is very much used for more mundane, terrestrial systems. A very common use of simulation is to understand whether a facility such as a restaurant or a train station or airport has enough resources such as tables or platforms or runways to satisfactorily serve the customers or travellers that might arrive into it.

As a concrete example, suppose we want to decide how many dining tables we should put in a restaurant. If we put too few tables, we will not be able to accommodate all customers who might want to eat in our restaurant. On the other hand, each table we put in has a cost. So we might want to determine, for each T where T is the number of tables we put, what our revenue is likely to be. Knowing the revenue and the cost of putting up tables, we should be able to choose the right value of T. To do this analysis, we of course need to know something about how many customers want to eat in our restaurant, and when. This of course is predictable only statistically. We will assume that we are given p, the probability that a customer arrives in any given minute of the "busy period" for restaurants, say 7pm to 10pm. Ideally, we should consider not single customers but a party consisting of several customers, and the possibility that a customer party might need more than one table. However, for simplicity we will assume that customers arrive individually and are seated individually at separate tables. On arrival, a customer occupies the table for some time, say during which he eats, and then leaves. Suppose that we are also given a function $e(t)$, that gives the probability that a customer eats for t minutes. For simplicity, suppose that the revenue is proportional to the total number of customers. Can we determine the total revenue for an arbitrary value of T, the number of tables we have? Note that an arriving customer will leave if all tables are occupied.

Problems such as this one can sometimes be solved analytically, i.e. we can write the expected revenue as a reasonably simple, easily evaluatable function of the different parameters. But this is often not possible if the probability model is complex. For example, in the above description, we implied that the eating time probability distribution $e(t)$ is a function only of t. But if there are many people in the restaurant, service might will be slower and each customer will occupy the table for longer periods. Thus perhaps the distribution should be a function of the number of customers present as well. In this case, it will be much more difficult to write down an analytical solution. In such cases, a common strategy is simulate the system. By this we mean the following. We pick random numbers from appropriate distributions to decide when customers arrive, how long they wait. Using this information we compute how many tables are occupied at each instant, which customers need to be turned away because the restaurant is full and so on.

In this chapter we will see how to perform this simulation. This simulation has a very different character from the cosmological simulation of Chapter 15. We will see that it is an example of a *Discrete Event Simulation*, which we consider at length in Section 21.1. We will develop some machinery to perform discrete event simulations using which we will perform the restaurant simulation. We will also consider a variation, which we will call the *coffee shack simulation*. The last topic in this chapter is the *shortest path problem* for graphs. We can get a fast algorithm for this abstract problem by viewing it as a simulation of a certain natural system. In Chapter 22 we develop a simulation of an airport, which uses the machinery we develop in this chapter, and then some.

21.1 Discrete event simulation overview

In principle, every simulation can be programmed in the manner of the cosmological simulation. In the cosmological simulation, for each time step we computed what happens to each star. Similarly we could compute what happens to each customer at each step, where a step again might be chosen to be a small enough time interval, say a minute. The total processing effort in this program organization is proportional to at least the product of the number of entities in the simulation (stars or customer parties) and the number of time steps. While this approach, often called the *time-driven* approach, is fine for the cosmological simulation, it is likely to be very inefficient for the restaurant simulation.

We explain the key idea with an example. Suppose we have some T tables, and the first T customers arrive (say that is how the random numbers got chosen) at time steps 1 through T. Clearly each of them will be given a table. Now, for each of these T customers we have (suitably randomly) fixed the time for which they will eat, and hence the time at which they will depart. Suppose among the departure times of these customers, and the arrival times of the remaining customers, the smallest number is t. Then we know that nothing of interest happens in our simulation between step $T+1$ to step t: the customers who were eating will keep eating. Thus we can directly jump to step t and process the departure or arrival whichever was supposed to happen then. This will clearly be more efficient than the time driven approach, in which we painfully examine each entity at each step.

In the new approach, we are essentially asking, "what is the earliest important event that will happen next?". The phrase "important event" is to be interpreted in the sense of an event that can change the state of some entity. If the earliest important event will happen at some time step t and the current time step is T, then we can directly jump to step t. The focus in this approach is on the important events in the system. Further, we use the term event in the sense of an instantaneous, or discrete, occurrence. Hence this approach is called the *Discrete Event Simulation* approach.

In general, a discrete event simulation works as follows. The system to be simulated consists of a set of active elements which we will refer to as *entities*, and some additional passive elements. Entities can be *awake* or *dormant*. When awake, an entity can examine and alter its own state or the state of the other entities, or the state of the passive elements. An entity may also cause a new entity to be created. These actions are the *events*, and they are deemed to happen instantaneously. After performing the required actions, an entity might decide to become dormant for some specified number of time steps. When that number of steps are deemed to have elapsed, then the entity must be woken up, and then it can perform more actions/events.

We will have a class `simEntity` whose instances will be the entities in the system. The class will have a member function `wakeup`. To wakeup an entity, we will merely invoke the `wakeup` function on it. We will maintain a set which will hold the entities when they are dormant. More precisely, our set will hold pairs of the form (t, e), where e is the dormant entity and t the time at which it needs to be woken up. So we will call this set the *Wakeup Requests Set* (WRS). Finally, we maintain a variable `time` which is used to hold the time till which we have simulated the system. At the start of the simulation, `time` is set to 0, and WRS is set to contain a (t, e) pair for each entity e which we know is either present at the beginning and wants to be woken up at time t, or is known to enter the system at time t.

The basic iteration of the simulation algorithm is as follows. Suppose we have simulated the system till some time t. Then the variable `time` will hold the value t. Let t' be the smallest time value that appears in any request in WRS at this stage of execution. Now clearly, nothing of interest will happen in our system between time t and t'. Hence we can safely set our variable `time` to t'. We then consider all entities that need to be woken up at t'. We select some entity e from these and invoke the `wakeup` function on it. As a part of `wakeup`, an entity may perform various actions, including examining and modifying program state. After finishing the actions required for the current `wakeup` call, an entity may decide it needs to become dormant again, till some time t''. So the last action in the function `wakeup` will typically be to insert a (t'', e) pair into WRS. If the entity has completely finished everything that it needs to do and wants to leave the simulation, it simply returns from `wakeup` without inserting anything into WRS. After this the basic iteration repeats. The simulation terminates when WRS is found to be empty.

Here is the definition of the class `simEntity`.

```
class simEntity{
 public:
  virtual void wakeup() = 0;
};
```

This class is abstract, i.e. it is not expected to be used directly to create instances, but instead, actual entities are expected to be instances of a subclass of `simEntity`. For example, in the restaurant simulation we will have a `Customer` subclass to represent customers. Or later, you will see that the entities will be aircraft and these will be instances of a `plane` class which will be a subclass of `simEntity`. As you will see, the superclass `simEntity` is useful because the code related to managing entities as they go to sleep and wake up can be written once, considering only objects of class `simEntity`, and this will get used for all subclasses. This is a big advantage of the object oriented organization.

The time entity pairs that we need will be represented using the `pair` class in STL (found in the header file `<util>`). We use the type `double` to represent time, so we need a pair of `double` and `simEntity`. Instead of putting `simEntity` into the pair, we put a pointer to it. Thus the class is obtained by writing pair<double,simEntity*>. The pair class is convenient because the comparison operators work on it: with the comparison happening lexicographically. Thus pair $(t, e) < (t', e')$ if $t < t'$ or if $t = t'$ and $e < e'$. Note that we are using pointers to entities and not the entities themselves. Comparisons are defined on pointers (they are treated as unsigned integers for this purpose). Thus instead of needing to demand "a pair (t, p) where t is smallest", we can merely demand "a smallest pair (t, p)".

Next we say how to represent the set WRS in which we will store these pairs. There is nothing special as far as the manner in which pairs get inserted into the set. However, removal from WRS happens in a very special manner: we always remove only a smallest pair from those stored, in the sense defined above. The priority_queue template class in STL (found in header file <queue>) supports exactly this mode of operation and is thus ideal for implementing WRS. We can obtain a class for prioirty queues of pair<double,simEntity*> by writing priority_queue<double,simEntity*>. Thus if we want to create an instance pq of this class we would write:

```
priority_queue<pair<double,simEntity*> > pq;
```

This is almost what we want, but not quite. For this definition, the method top will return the *largest* pair in the queue. But we can change this behaviour so that it instead returns the smallest. To change the default behaviour, we need to know the prototype for priority_queue, which is as follows.

```
template<class T, class C = vector<T>, class cmp = less<typename
C::value_type> >  priority_queue;
```

A prototype of a template is similar to a function prototype: both give the arguments and their types, and possible default values. In this case, the priority_queue template takes 3 arguments. The third argument cmp is used to decide what to return. It defaults to less, which is simply the operator <. Thus a priority queue returns that element x such that there is no y such that x < y. Thus a largest element is returned. To get a smallest instead, we must make the third argument be the > operator, and this is specified as greater<pair<double,simEntity*> > in our case. But to specify a non-default value for the third argument, we must also specify a value for the second. Thus we define the class as shown.

```
class simQueue {   // ************* Implementation version 1 *************
  double time;
  priority_queue<pair<double,simEntity*>, vector<pair<double,simEntity*> >,
    greater<pair<double,simEntity*> > > pq;
 public:
  simQueue(){time=0;}
  void insert(double sleepTime, simEntity *pP){
    pq.push(make_pair(time+sleepTime,pP)); // wakeup at current time + sleepTime
  }
  void process_till_empty(){
    while(!pq.empty()){
      pair<double,simEntity*> sqe = pq.top();
      pq.pop();
      time =sqe.first;
      sqe.second->wakeup();
    }
  }
  ostream & log(){
    cout << time << ") ";
    return cout;
  }
};
```

WRS can be represented using an instance of simQueue.

The first method supported by simQueue is insert, for inserting a wakeup request. We compose a pair from the wakeup time and the entity pointer using the make_pair provided in the pair template class. The pair is then inserted into the priority queue.

The method process_till_empty does as it says, it repeatedly picks the smallest element in the queue and wakeups the entity. The instance variable time is updated. Note that the smallest element in the queue can be removed by using the pop method, or we can just examine it using the top method.

The last method, log, is for reporting convenience. It is used to print messages to the screen, but each message is prefaced by the current time. Note that a reference to the console output, cout is returned, so that the rest of the message can be appended using the << operator, as will be seen in the next section.

This is a perfectly adequate representation. However, we will give an alternate implementation which we will use in the rest of the chapter. As you will see this is slightly more convenient. The new implementation is based on the observation that in any simulation we will have only one WRS. So, we implement WRS using static variables in the class simQueue as shown.

```
class simQueue {                        // ** Real Implementation **
  static double time;
  static priority_queue<pair<double,simEntity*>,
    vector<pair<double,simEntity*> >,
    greater<pair<double,simEntity*> > > pq;
 public:
  void insert(double sleepTime, simEntity *pP){  /* as before */ };
  static void process_till_empty(){              /* as before */ };
  static ostream & log(){                        /* as before */ };
};

double simQueue::time = 0;
priority_queue<pair<double,simEntity*>, vector<pair<double,simEntity*> >,
    greater<pair<double,simEntity*> > > simQueue::pq;
```

The implementation of the functions insert, process_till_empty and log is as before, so that is not repeated. Note the definition of the static variables simQueue::time and simQueue::pq at the end. Static variables only get declared when they are mentioned in the class declaration, and must be defined outside the class declaration, as shown.

For the new implementation, we do not need to instantiate the class simQueue. We already have the set WRS represented. We can insert into it using the function simQueue::insert and to process the inserted pairs we call simQueue::process_till_empty. To print messages prefixed by the current simulation time we call simQueue::log.

21.2 The restaurant simulation

Suppose for the sake of definiteness that we have a restaurant with 5 tables, in which at each minute between 7pm and 10pm a customer arrives with probability 1/10. Suppose that the eating time for a customer is in the range 21-40 minutes, with all durations equally likely. This is the system we want to simulate.

We will use the simEntity and simQueue classes given above. We will represent customers using a Customer class. This will be a subclass of simEntity.

The main program is given first. It begins by creating a restaurant object in which we hold the information about the number of tables, the number that is occupied, and the customer arrival probability. Then for each t, where t represents each of the 180 minutes between 7 and 10 pm, a customer is generated with the required probability. The time that each customer spends eating is generated to be a random number between 21 and 40. For generating random numbers, we use the function randuv from Section 8.6.1. The customer object if created is inserted into simQueue, to be woken up at time t.

```
struct Restaurant{
  const int capacity;   // number of tables in the restaurant
  int nOccupied ;    // number of tables occupied
  double arrivalP;
  Restaurant(int c, double ap) : capacity(c), arrivalP(ap) { nOccupied = 0; }
};

int main(){
  Restaurant restaurant(5,0.1); // restaurant has 5 tables,
                                // arrival probability = 0.1
  int customer_id = 0;
  for(int t=0; t<180; t++)
    if(randuv(0,1) <= restaurant.arrivalP){
      double eatingT = randuv(21,40); // uniform between 21,40
      simQueue::insert(t, new customer(++customer_id, eatingT,
                                       &restaurant));
      }
  simQueue::process_till_empty();
}
```

Next we present the definition of the Customer class. The main part in this is the action the customer takes when woken up. A customer is woken up first when time becomes equal to the time at which the customer is supposed to arrive, i.e. the time for which it is created in the above code. When this call to wakeup happens, the actions of the customer as he arrives at the restaurant must be mimicked. On arrival the customer can enter the restaurant if there is an unoccupied table. If so, the customer enters and will start eating immediately, in our simplistic model. After the eating time eatingT elapses, the customer must leave the restaurant. During the eating process the state of the customer does not change, and so for the purposes of the simulation the customer can be thought of as becoming dormant. Thus if the customer does find a table, then the count of occupied tables is incremented, and the customer is inserted back into WRS. At this point the eating time is specified as the duration for which the customer will be dormant. When this duration elapses, the wakeup function is again called. This time the wakeup function must merely print out a message that the customer is leaving the restaurant.

Thus for each customer, there can be two calls to the function wakeup, on arrival and after finishing eating. To distinguish the two cases, in each Customer object we will have a data member state. Depending on the value of state, the function wakeup will execute appropriate code. We use the value 0 to indicate that a customer is just entering and a 1 to indicate that a customer is eating.

```
class Customer : public simEntity{
  int id;
  double eatingT;
  int state;
  Restaurant pRest;
public:
  customer(int i, double t, Restaurant *ptr) : id(i), eatingT(t), pRest(ptr) {
    state = 0;                                // initial state = about to enter
  }
  void wakeup(){
    switch (state){
    case 0:                                   // if about to enter
      if(pRest->nOccupied >= pRest->capacity){
        simQueue::log() << " Number of occupied tables: " << pRest->nOccupied
                        << "  Customer " << id << " disappointed.\n";
      }
      else{
        simQueue::log() << " Number of occupied tables: " << pRest->nOccupied
                        << "  Customer " << id << " entered.\n";
        ++pRest->nOccupied;
        state = 1;                            // set state to eating.
        simQueue::insert(eatingT,this);
      }
      return;
    case 1:                                   // if was eating, which ended.
      simQueue::log() << " Number of occupied tables: " << pRest->nOccupied
                      << " Customer " << id << " finishes.\n";
      --pRest->nOccupied;
      return;
    }
  }
};
```

21.3 Simulating a coffee shack

Consider now, a roadside coffee shack manned by a single server. Suppose the shack serves beverages and food, all of which require some effort and time from the server. If a customer arrives while the server is busy with a previous customer, then the new customer must wait. So a line of waiting customers might form at popular coffee shacks. Given the probability of customer arrival and the probability distribution of the service time, can we predict how much business the shack gets and also how long the line becomes?

Now we need to model a simulation entity (customer) waiting for a resource (server's attention) to become available. One way to deal with this is so called *busy waiting*: periodically (say every minute) the entity wakes up to check if the resource has become available. It is more efficient, instead, if a departing customer wakes up the next customer in the queue so that the next customer can be served. This is the idea we will implement.

21.3.1 A `Resource` class

The key notion is that of a `Resource` class. Customers try to reserve the resource, and if it is in use, they wait in a queue. For this we will use the `queue` class in STL. We will implement a resource which keep tracks of who (which `simEntity`) is using it, in an instance variable (`owner`). We could have merely kept track of whether the resource is in use or not by using a boolean instance variable; knowing who is using the resource will make this class more useful.

A `simEntity` can attempt to acquire the resource by calling the `reserve` method. This returns `true` if the resource can be reserved for this `simEntity`, or is already reserved by this `simEntity`. Otherwise, `false` is returned. If a resource cannot be reserved, a `simEntity` may choose to `await` its availability. This is implemented simply by putting the `simEntity` on the queue associated with the resource. Finally, a resource can be `release`d. Release is implemented as follows. If no entity is waiting, we merely mark the `owner` of the resource to be `NULL`. If some entity is waiting, then we make the entity at the head of the queue be the `owner`. We also put that entity into `simQueue` with a delay of 0 so that it will be woken up in the current step itself.

```
class Resource{
  queue<simEntity*> q;
  simEntity* owner;
public:
  Resource(){owner = NULL;};
  int size(){ return q.size(); }
  bool reserve(simEntity* pS){
    if (owner == NULL)
```

```
      owner = pS;
    return owner == pS;
  }
  void await(simEntity* pS){
    q.push(pS);  // should be called only if reserve fails.
  }
  void release(){
    if(!q.empty()){
      owner = q.front();
      q.pop();
      simQueue::insert(0,owner);
    }
    else owner = NULL;
  }
};
```

21.3.2 The simulation

Using the Resource class, the simulation is easily written. The main program for simulating a 60 minute duration is as follows.

```
int main(){
  const float arrivalP = 0.15, minServiceT=3, maxServiceT=9;
  int cid = 0;
  Resource server;

  for(int t=0; t<60; t++)           // 60 minute duration
    if(randuv(0,1) <= arrivalP){
      double serviceT = randuv(minServiceT, maxServiceT);
      simQueue::insert(t, new Snacker(++cid, serviceT, server));
    }
  simQueue::process_till_empty();
}
```

The general outline is as before. We create a Resource to model the server, which is called server in the code. Then for each minute we generate a customer, with the arrival probability arrivalP. The customer in this case is an instance of the class Snacker which we describe below.

```
class Snacker : public simEntity{
public:
  int id;
  int serviceT;
  int state;
  Resource &server;
  Snacker(int i, int t, Resource &s) : id(i), serviceT(t), server(s){
    state = 0;
  }
  void wakeup(){
    switch (state){
    case 0 :
      ++state;
      simQueue::log() << " Customer " << id << " in queue.\n";
      if(!server.reserve(this)) server.await(this);
      else simQueue::insert(0,this);
      break;
    case 1 :
      ++state;
      simQueue::log() << " Customer " << id << " being served.\n";
      simQueue::insert(serviceT,this);
      break;
    case 2 :
      simQueue::log() << " Customer: " << id << " finishes.\n";
      server.release();
    }
  }
};
```

We first remark about the constructor for `Snacker`. Note that the argument `s` to the constructor is a reference to the server, and not a pointer to the server. Similarly, the member `server` is also a reference variable (Section 11.2.2). Thus we can save the reference in the reference variable, and use it during the lifetime of `Snacker`.[1]

The behaviour of the `Snacker` is more complex than that of the `Customer`. The `Snacker` will potentially be woken up thrice: first on arrival, second on getting access to the `server`, and finally when the service finishes. So we need a `state` variable which will take values 0,1,2. Based on this variable, appropriate code will be executed.

`state 0` This is the case representing arrival of the snacker into the coffee shack. The snacker tries to reserve the server. If the server can be reserved, then the snacker is ready to start being served, which happens in the next state. For this, the snacker puts itself back into `simQueue` with sleep time of 0. If the server cannot be reserved, then the customer must wait for the resource. When the resource becomes available, then the code in `Resource` will cause the snacker to be put into `simQueue` for the next step.

`state 1` This is the case when the snacker has got access to the server. The action in this case is simple, we must model the snacker being served. For this the snacker puts itself back into `simQueue` to be woken up after its service time, `serviceT`.

`state 2` This is the case in which the service has finished. So the snacker just leaves, i.e. nothing is put on `simQueue`.

[1] Analogous to what we did in the restaurant simulation, we could have only used pointers. We are using reference variables just to show how reference variables can be used.

21.4 Single source shortest path

The shortest path problem we considered in Section 13.6 was the *all-source-shortest-path* problem, so called because we wanted to find the lengths of the shortest paths from *all* cities to all other cities. We will use the term `distance` to denote the length of the shortest paths. In this section we consider the *single-source-shortest-path* problem, i.e. we want to know the distances from just one of the cities, to all other cities. As in Section 13.6, we will focus on the problem of finding the distances, the paths themselves can be identified with a little additional book-keeping, which is left for the Exercises. The algorithm we discuss here, attributed to Edsgar Dijkstra, is much faster than the algorithm of Section 13.6. So clearly this algorithm is more suitable if you want the distances from just one city, which will be referred to as the *source*, in what follows.

Dijkstra's algorithm can be viewed as a computer analogue of the following physical experiment you could undertake to find the distances. For the experiment we need many cyclists who can ride at some constant speed, say 1 km/minute. Specifically, we need to have as many cyclists in each city as there are roads leading out of it. If we do have such cyclists, here is how they could find the length of the shortest paths.

To start with, all the cyclists assemble in their respective cities. Each cyclist is assigned one road leading out of the city, and the job of the cyclist will be to travel on that road when asked. Thus at the beginning, for each road in our map, we have a cyclist waiting.

At some time which we will call 0, the cyclists in the source city start pedalling. At time 0 the cyclists in the other cities do nothing. As an example, suppose our graph is the map of Figure 13.1, and we want the distances from Nashik. So at time 0, three cyclists start pedalling from Nashik to respectively Nagpur, Mumbai and Pune.

Here is what happens when a cyclist reaches her destination. If she is the first person to reach that city, then she signals the cyclists waiting in the city to start pedalling. If she is not the first cyclist to arrive into the city, i.e. someone arrived earlier, she does nothing. Continuing our example, the cyclist from Nashik would arrive at time 200 into Mumbai, where we are measuring time in minutes from the start. She would be the first one to arrive there, so she would flag off the 3 Mumbai cyclists who would then start travelling towards Kolhapur, Pune, and Nashik respectively. Of these 3 the cyclist heading to Pune would reach 160 minutes later, at time 360. However, when she reaches Pune, she would have found that the cyclist from Nashik has already arrived at time 220. So the cyclist arriving from Mumbai into Pune would need to do nothing.

The experiment ends when all the cyclists have finished their journey.

We will show that: (a) the length of the shortest path from the source to any city is simply the time in minutes when the earliest cyclist arrives in that city! (b) we can use discrete event simulation to simulate this system.

We explain (a) first. Let S denote the source city, and C be any city. Let t be the time at which the first cyclist arrives into C. We argue that there must be a path from S to C of precisely this length. To see this, consider the cyclist that arrives into C. We follow this cyclist backward in time to the city from which he started. There, he was flagged off by some other cyclist, whom we follow back in time, and so on. Eventually, we must reach the city S, at time 0. In this process, note that we are not only going back in time but also continuously travelling back, at 1 km/minute. Thus, we must have covered, backwards, exactly the same distance as the time taken. Thus we have proved that there exists a path from S to C of length equal to the time at which the first cyclist arrives in C. We now prove that it is the shortest.

Consider a shortest path P from S to C, the cities on it being $c_0, c_1, \ldots, c_k$ in order, with $c_0 = S$, and $c_k = C$. Let d_i be the distance from c_0 to c_i along the path. We will prove that the first cyclist leaves c_i latest at time d_i, for all i. Clearly, this is true for $i = 0$: indeed a cyclist leaves c_0 at $0 = d_0$. So assume by induction that a cyclist leaves c_i at d_i or before. But this cyclist travels at 1 km/minute, and requires $d_{i+1} - d_i$ time to travel from c_i to c_{i+1}. Hence he will arrive at c_{i+1} at time at most $d_i + d_{i+1} - d_i = d_{i+1}$. Thus the induction is complete. Thus we know that some cyclist must arrive at $c_k = C$ at time at most the length of a shortest path P. But we proved that the time of arrival must equal the length of some path. Hence it follows that the first cyclist arrives at time exactly equal to the length of the shortest path.

We next show that our algorithm can be programmed as a discrete event simulation.

21.4.1 Dijkstra's algorithm as a simulation

The first question, of course is how to represent the graph. We could use the same representation as in Section 13.6. We use a different representation, shown in Figure 21.1 similar to the representation for trees discussed earlier.

The main class is `Graph` which holds a vector, `vertices`, the `i`th element of which is an object of class `vertex` containing information about the `i`th vertex. Each vertex object contains a vector `edges` which stores information about the edges leaving that vertex. Suppose `G` is a `Graph`. Then `G.vertices[i].edges[j]` stores information about the `j`th edge leaving vertex `i`, specifically it holds the following: (a) a pointer `vptr` to the vertex which is the other endpoint of this edge, (b) a double `length` giving the length of this edge. The member `arrivalT` in each `vertex` object is meant for storing the time at which the first cyclist arrives into that vertex. Note that `length` and `arrivalT` are needed specifically for our simulation; if you want to develop other graph algorithms, then you would not have these members but possibly some other members.

The class *Graph* contains a constructor which can read in the graph from the file whose name is given as an argument. Figure 21.2 shows a sample input file. This file represents the graph of Figure 13.1. The first number in the file gives the number of vertices. The constructor sets the size of the array `vertices` to this number. This will cause elements of the vector `vertices` to be created. Thus a `vertex` object is created for each vertex in the graph. Note the constructor for `vertex`: it sets `arrivalT` to `HUGE_VAL`, which represents ∞. This serves to denote that as of now, the member `arrivalT` is undefined. Next, the constructor reads information about edges in the graph. This consists of triples `v1, v2, dist`, where `v1`, `v2` give the endpoints of the edges, and `dist` gives the distance between the endpoints. We must store the information about this edge in the structure `vertices[v1]` which stores information related to vertex `v1`, as well as in `vertices[v2]` which stores information related to vertex `v2`. That is done in the two statements in the loop. When the loop finishes, the graph will have been constructed. We will discuss the `wakeup` member function later.

Note that the structure `vertex` contains a vector of `edge` objects. Thus we must define

```
struct vertex;   // forward declaration, not definition.
struct edge{
  vertex* vptr;
  double length;
  edge(vertex* vp, double d){vptr = vp; length = d;}
};

struct vertex : public simEntity{
  vector<edge> edges;
  double arrivalT;
  vertex(){arrivalT = HUGE_VAL;}
  void wakeup(){
    if(arrivalT > simQueue::getTime()){
      arrivalT = simQueue::getTime();
      for(int i=0; i<edges.size(); i++){
        simQueue::insert(edges[i].length, edges[i].vptr);
      }
    }
  }
};

struct Graph{
  vector<vertex> vertices;
  Graph(char* infilename) {
    ifstream infile(infilename);
    int n;
    infile >> n;
    vertices.resize(n);
    double dist;
    int end1, end2;
    while(infile >> end1){
      infile >> end2 >> dist;
      vertices[end1].edges.push_back(edge(&vertices[end2],dist));
      vertices[end2].edges.push_back(edge(&vertices[end1],dist));
    }
  }
};
```

Figure 21.1: Graph representation

File content	Explanation
6	Number of cities
0 1 450	Kolhapur Mumbai distance
0 5 350	Kolhapur Satara distance
1 2 160	Mumbai Pune distance
1 3 200	Mumbai Nashik distance
2 3 220	Pune Nashik distance
3 4 500	Nashik Nagpur distance
5 2 50	Satara Pune distance

Figure 21.2: Input file for graph of Figure 13.1

the class `edge` before defining the class `vertex`. However, the structure `edge` contains a pointer to a `vertex` object. This might seem to require that we define `vertex` before `edge`. This is not true, since `edge` only contains a pointer to `vertex`, it suffices if `vertex` is *declared* before `edge`. This is done by the first line of Figure 21.1.

The main program creates the graph, and starts of the simulation of the movement of the cyclists, as we explain below.

```
int main(int argc, char** argv){
  Graph G(argv[1]);  //  argv[1] = name of file from which to read graph
  int source;
  stringstream(argv[2]) >> source;  // index of source city

  G.vertices[source].wakeup();      // Flag off the cyclists in source
  simQueue::process_till_empty();   // Simulate until all cyclists finish.

  for(int i=0; i<G.vertices.size(); i++) // print all arrival times
    cout << G.vertices[i].arrivalT << " ";
  cout << endl;
}
```

The program uses command line arguments. The first command line argument `argv[1]` gives the name of the file which contains data to build the graph. We supply this file name to a constructor of the class `Graph` which builds a graph object `G` for us. The second command line argument, `argv[2]` is expected to be an integer, and it gives the index of the source node. For this we first convert the string `argv[2]` to a `stringstream`, and then read from it. For this we need to include the header `<stringstream>`. Then we start off the simulation.

The important event in the simulation is the arrival of a cyclist into a city. These are the only events in our simulation; `wakeup` does whatever is supposed to happen when a cyclist arrives. The arrival of a cyclist into city `i` corresponds to execution of `G.vertices[i].wakeup()`. When a cyclist arrives, the arriving cyclist must check if she is the first to arrive. Correspondingly, the function `wakeup` as implemented in the class `vertex` in Figure 21.1 does the following. It checks whether the member `arrivalT` is `HUGE_VAL`. Note that `arrivalT` was set to `HUGE_VAL` when the vertex was created, and is changed only during the execution of wakeup. Thus if arrivalT still equals HUGE_VAL, then this cyclist is the first to arrive, and it must do the following:

1. Record the correct arrival time into arrivalT.

2. The cyclists must be flagged off to leave the current vertex. A cyclist must be flagged of for each neighbouring city i. The cyclist will reach the corresponding city, pointed to by edges[i].vptr, after covering the distance edges[i].length, i.e. after that much time. Hence, Hence we insert a request in simQueue to wakeup *(edges[i].vptr) after edges[i].length minutes from the current time.

If arrivalT is not HUGE_VAL, then it must have been set to a finite value in some previous wakeup call, i.e. when some cyclist visited earlier. In that case the current cyclist must do nothing.

To start off the simulation, we must flag off the cyclists in the source vertex. The member function wakeup does precisely this, and hence this is what is called by main. After that, main merely waits for the simulation queue to be empty, i.e. for all cyclists to finish their journey. Finally the distances to each vertex i from the vertex source as computed in G.vertices[i].distance are printed.

Exercises

1. Modify the restaurant simulation to report how many customers left disappointed, how long after the closing time did the customers stay around, the number of customers in the restaurant on the average.

2. Generalize the coffee shack problem so that there are several servers. This is also like adding a waiting room to the restaurant. You will need to modify resource. Generalize the class so that at most some k clients can be using the resource simultaneously. You may find it easier to do this if you do not keep track of which clients are using the resource, but just keep track of how many clients are using the resource.

3. Suppose every minute a customer enters a store with a probability p. Suppose that on the average each customer spends t minutes in the store. Then on the average, how many customers will you expect to see in the store? *Little's law* from queueing theory says that this number will be pt. Modify the coffee shack simulation and verify Little's law experimentally. The law requires that no customers are turned away, and that the average is taken over a long (really infinite) time. So you should remove the capacity checks, and run the simulation for relatively long durations to check. More code will be needed to make all the measurements.

4. Write a simulation of a restaurant in which customers can arrive in a group, rather than individually. Suppose a group can have upto 5 members, all sizes equally likely. Suppose further that tables in the restaurant can accommodate 4 customeres, so if a party of 5 arrives, then two adjacent tables must be allocated. Thus, the party must wait if two adjacent free tables are not available. Write a simulation of such a restaurant. Assume that the tables are in a single line, so tables $i, i + 1$ are adjacent.

You will have to decide on how a table will be allocated if several tables are free: this will affect how quickly you serve parties of 5 members.

5. Have an additional command line argument which gives the index of a *destination* city, for the shortest path program. Modify the program so that it prints the shortest path from the source to the destination city, as a sequence of the numbers of the cities on the way. Basically, in each `vertex` you must store information about where the first cyclist arrived from. This will enable you to figure out how the shortest path arrives into a vertex, recursively. This requires a somewhat significant modification. Define a `cyclist` class which is a `simEntity`, rather than making a vertex a `simEntity`. The cyclist objects should contain information about which cities they travel between. Now when a cyclist arrives into a city, she will know where she arrived from.

6. Modify the shortest path algorithm to use city names instead of city numbers in the input file.

7. Build a simulator for a circuit built using logic gates. Consider the gates described in Exercise 15 of Chapter 5. You should allow the user to build the circuit on the graphics window. You should also allow a delay δ to be entered for each gate. A gate takes as input values 1 or 0, and produces output values according to its function. However, the output value is reliably available only after its delay. Specifically, suppose some input value changes at time t. Suppose this will cause the output value to change. Then the new correct value will appear at the output only at time $t + \delta$. During the period from t to $t + \delta$ the value at the output will be undefined. For this you should use the value `NAN` supported as a part of the header file `<cmath>`. The value `NAN` represents "undefined value", actually the name is an acronym for "Not A Number". This value behaves as you might expect: do any arithmetic with it and the result is `NAN`.

Chapter 22

Simulation of an airport

Suppose there are complaints about efficiency of an airport in your city: say flights get delayed a lot. Is it possible to pinpoint the reason? Is it then possible to state the best cure to the problem: that you need to build an extra runway, or some extra gates, or perhaps just build a completely new, bigger airport? A simulation of the airport and how it handles aircraft traffic can very much help in making such decisions.

The simulation will take as input information about the runways and other facilities on the airport, and about the aircraft arriving into the airport from the rest of the world. It will then determine what happens to the aircraft as they move through the airport, what delays they face at different points. The average of these delays is perhaps an indicator of the efficiency of the airport. To answer questions such as: how much will an extra gate (or runway or whatever) help, you simply build another simulation in which the extra gate is present, and calculate the average delay for the new configuration. In addition to textually describing what happens to each aircraft as it progresses through the airport, it is also desirable to show a graphical animation in which we can see the aircraft landing, taxiing or waiting at gates. An animation is possibly easier to grasp – perhaps seeing the aircraft as they move might directly reveal what the bottlenecks are.

The first step in building a simulation is to make a computer model of the relevant aspects of the system being simulated. When you make a computer model, or a mathematical model, of any entity, doubtless you have to throw away many details. A trivial example: the colour of the airport building is irrelevant as far as its ability to handle traffic, so that may be ignored in our simulation. On the other hand, the number of runways in the airport is of prime importance, and so cannot be ignored. Other factors that perhaps cannot be ignored include the number of gates at which aircraft can park to take in and discharge passengers, the layout of the taxiways that connect the runways and the terminals. Other factors that are perhaps less important are the placement of auxiliary services (e.g. aircraft hangars) and traffic associated with these services and how it might interfere with aircraft movements. In general, the more details you incorporate into your model, the more accurate it is likely to be. However, models with relatively few details might also be useful, if the details are chosen carefully.

In this chapter, we will design a program to simulate a fairly simple airport. We will begin by describing the airport we want to simulate and the rules under which the airport operates. Then we give the general structure of a possible implementation. An important

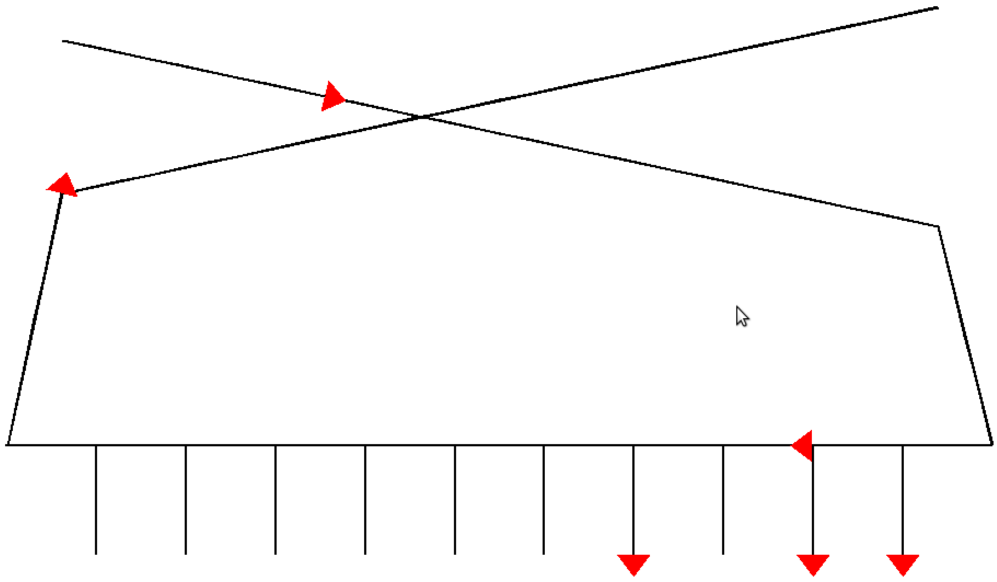


Figure 22.1: Airport layout with planes

problem in simulating complex systems such as an airport is *deadlock*. We discuss how deadlocks can be dealt with in real life and in programs.

22.1 Airport configuration and operation

The configuration of our airport shown in Figure 22. In our model of the airport, we will only consider the runways, taxiways, and gates for simplicity. The two crossing lines at the top are two runways. The other lines are taxiways. The long horizontal line at the bottom is the main taxiway, and the nearly vertical segments on the sides we will refer to as the left and right taxiways respectively. There are branches going off the main taxiway to the gates. We have not shown the gates, but they are supposed to be present at the end of these short branches. So in this airport there are meant to be 10 gates, which we will number 0 through 9, right to left. The small triangles are meant to represent aircraft. As you can see there are three aircraft waiting, at gates 0, 1, and 3, and three others on the runway and taxiways. If you ignore the branch taxiways, the runways and the other taxiways constitute a single long path, starting in the top left corner, running clockwise over itself to end in the top right corner. We will call this the *main path*. Indeed, for simplicity, we will require that the main path be used in the clockwise direction. Thus the runway starting at the top is the landing runway and the runway ending at the top right is the takeoff runway. The branch taxiways going to the gates are expected to be used in both directions.

Our configuration is rather simplistic, except for the intersecting runways. Intersecting runways are not rare, by the way – in fact the Mumbai airport has intersecting runways, which is our inspiration for including them. But of course both the runways in Mumbai can be used for takeoffs as well as landings, and the taxiways and gate placements are more elaborate.

At a high level, the operation of an airport can be described as follows. Each aircraft lands and taxies to a gate. The aircraft then waits at the gate for a certain *service* time. After that the aircraft taxies to the runway and takes off. This entire process has to be controlled by the airport authorities so as to ensure safety and efficiency.

22.1.1 Safe operation rules

The gist of the safety requirements is: aircraft movement should be planned so that at all times aircraft are well separated from each other. A certain minimum separation is required even as aircraft are taxiing. The separation between aircraft must be larger when they are travelling at high speeds, as will be the case when they are landing or taking off. So we will in fact require that there be at most one aircraft on each runway at any instant. But we need an even stronger *half-runway-exclusion rule* because our runways overlap. As shown, the runways intersect in the initial portion, so we will further require that if the initial half of the take off runway contains an aircraft then there should be no aircraft in the initial half of the landing runway, and vice versa.

22.1.2 Scheduling strategy

The exact schedule according to which aircraft land and takeoff and even move around while on the airport is decided by the air traffic controllers at the airport. They must obey the safe operation rules and in addition resolve conflicting requests. For example, if two aircraft request permission to use the runway (either for take off or for landing) at the same time, then permission can be granted to only one. This decision will have to be taken by the air traffic controllers. Such decisions will be made so as to acheive certain goals, e.g. say to minimize the average delay, or some weighted average delay with the weights being the priorities of the different aircraft. Another issue concerns gate allocation. When an aircraft arrives it must be assigned a gate at which it is to wait. In general, each aircraft may have its preferred gates at which it would like to wait. In order to perform a simulation we need to know the precise scheduling strategy and gate assignment protocol used by the airport.

For our simulation we will use a very simple *first come first served* scheduling strategy. Basically, we will assume that each aircraft requests permission from the traffic controller for each action it needs to perform, just as it becomes ready to perform the action. If several aircraft ask permissions to perform actions which require a common resource (say the runway), then permission is granted to the aircraft which asked earliest, and the other aircraft must wait. Of course many other strategies are possible. For example, we might decide to give higher priority to landings than takeoffs because it is easier for a plane to wait on ground that wait midair![1] This is explored in an exercise.

[1] An aircraft must begin its descent much earlier than its landing time, and once the descent has begun, the landing cannot be postponed in normal circumstances. However our first come first serve strategy may require a flight arrival to be delayed. So to make this more realistic, we can assume that we are given

As to gate allocation, we will assume that all aircraft can wait at all gates, and say the least numbered free gate will be allocated.

22.1.3 Simulator input and output

The input to the simulator is of two kinds. First, we are given the times required by an aircraft to traverse each segment of the taxiway and the runways. This assumes that the times are identical for all aircraft, and this is of course a simplification. Next, we are given the data about arriving aircraft. For each aircraft, we are given the arrival time, and the service time, i.e. the amount of time the aircraft needs to wait at a gate.

The primary output from the simulator will be: (a) an animation of the aircraft as they enter the airport, move to a gate, halt for the required time, and then take off and leave, (b) a text record of the times at which these events happen. When designing an animation, we need to decide how frequently will we show the state of our airport. Do we show it every second, or every minute, or only when something interesting happens, e.g. an aircraft arrives or leaves or stops at its gate? For simplicity, we wil assume the state is to be shown after every unit time interval, whatever the unit time we define in the program.

In addition, we may require several derived outputs. Let us define the delay of an aircraft to be the additional time it spent over and above when it could have departed had the airport been completely empty. So we might be required to compute the average delay. Such analyses and extensions are left to the Exercises.

22.2 Implementation overview

We can use either a time driven or an event driven approach. Since we are expected to show what happens at each instant of the simulation, a time driven approach might appear suitable, and the Exercises ask you to build a simulation using this approach. Note however that at each instant only very few aircraft will be active. So the event driven approach will also be convenient. This is what we use here. In fact, we will directly use the `simEntity` and `simQueue` classes we developed for the restaurant simulation. The entities will be of course be the aircraft. We will have a `plane` class to represent aircraft. Our simulation will in fact substantially resemble the restaurant or coffee shack simulations from Chapter 21. Just as customers moved through the restaurant or the coffee shop acquiring and releasing resources and waiting, so will the planes. We will not explicitly represent air-traffic controllers, instead the scheduling will be done by the code in the `plane` class. By suitably creating resources which the planes must reserve, we will have the effect of the controllers permitting the planes to move and allocate gates, and of course enforce safe operation rules.

Here is how we will enforce safe operation rules. The main idea is to break runways and taxiways into segments and make each segment a `resource` (Section 21.3). An aircraft must reserve a segment before moving on it. This way there can be at most one aircraft on each segment, and thus by choosing sufficiently long segments we can keep the aircraft well separated. The division into segments will be as follows. The two runways will be separate

the landing times well in advance, so that the aircraft can actually delay the arrival to suit our computed schedule if needed.

segments, and so will the the left and right taxiways. The main taxiway will be broken up into segments at the points where the branch taxiways leave from it. Since there are 10 gates, the main horizontal taxiway will be split into 11 segments. The branch taxiways will constitute separate segments by themselves. We will construct a taxiway class which will represent taxiways. This class will inherit from the resource class so that it can act like a resource.

To implement half-runway-exclusion we will use the following trick. Whenever a plane needs to land or take off, we will require it to reserve a fictitious rwCommon taxiway in addition to reserving the landing or takeoff runways respectively. Since only one plane can reserve any taxiway, this will ensure that only one plane can take off or land at the same time. After a plane has landed and traversed half the runway, we want to allow another plane to start taking off. To enable this, we simply release rwCommon as soon as the plane gets to the middle of the landing runway! Same thing for a plane taking off – it will also release rwCommon when it gets to the middle of the takeoff runway.

We will not represent the gates explicitly. We will model a plane waiting at gate i by having it wait at the end of branch taxiway i. The plane will traverse this taxiway, go to the end and wait for its service duration. During this period as well as during the period that it goes back to the main taxiway it will not release its reservation on branch taxiway i. This models the constraint that two planes will not use the same gate at the same time. To allocate a gate we merely examine all the branch taxiways and determining if any is free, and if so reserve it. This action takes place when the requesting plane is on the first segment of the main taxiway.

All the reservation actions happen as a part of the wakeup method of the plane class, just as all the simulation logic in the restaurant and coffee shack simulation was a part of the wakeup method in the customer class (Sections 21.2 and 21.3).

22.2.1 Main program and main data structure

The main data structure in the program is a vector of all taxiways, and the taxiway rwCommon. These are created by the main program. The main program also reads in the arrival and service times of the planes, and creates corresponding plane objects. These objects are inserted into simQueue, to be woken up at their arrival times. After this we let the simulation unfold itself by calling simQueue::process_till_empty.

```
vector<taxiway*> TAXIWAYS;
taxiway *rwCommon;

int main(){
  initCanvas("Airport Simulator",0,0,1000,1000);
  configure_taxiways_and_runways();
  initialize_sq_with_arriving_planes();
  simQueue::process_till_empty();      // current time is 0.

  getClick();
  closeCanvas();
}
```

The function configure_taxiways_and_runways sets up the data structures to represent taxiways and runways. We discuss this function in the next section, which also discusses the taxiway class in detail.

The function initialize_sq_with_arriving_planes creates the planes, as given below. To use simQueue we must of course include the header file sim.h from the last chapter, and also use sim.o while compiling.

```
void initialize_sq_with_arriving_planes(){
  ifstream arFile("arrivals.txt");

  int arrivalT, serviceT;
  int id = 1;                 // Planes are assigned numbers.
  while(arFile >> arrivalT){
    arFile >> serviceT;
    plane *p = new plane(id++,arrivalT,serviceT);
    simQueue::insert(arrivalT,p);
  }
}
```

The plane class is discussed later.

22.3 The taxiway class

Instances of the taxiway class must serve two purposes: they must be visible on the screen as lines, and the planes must be able to reserve them. So it is natural to derive the taxiway class from the Line class and the resource class of the preceding chapter.

```
class taxiway : public Line, public resource{
public:
  int traversalT;
  double stepsize;
  taxiway(float xa, float ya, float xb, float yb, int trT)
    : Line(xa,ya,xb,yb), traversalT(trT)
  {
    stepsize = sqrt(pow(xa-xb,2)+pow(ya-yb,2))/traversalT;
  }
};
```

The taxiway constructor first creates the Line representing the taxiway on the screen. Ideally we should distinguish the on-screen line from the real taxiway, and provide details about the real taxiway separately. For simplicity we have assumed that the on-screen taxiway and the real taxiway will have same coordinates on the screen as well as the ground (say the units have been conveniently selected). In constructing a taxiway we also provide the time required to traverse it in some hypothetical time units. Since we know the length of the taxiway we calculate how much an aircraft moves forward each (hypothetical) step when on this taxiway – this information is needed to perform the animation.

Note that the resource constructor is not explicitly called, so a call with no arguments will be inserted by the compiler. This will set the owner of the taxiway (derived from resource) to NULL, indicating that initially the taxiway is unreserved.

The function configure_taxiways_and_runways will instantiate taxiways to create the main path and the branch taxiways.

The segments of the main path will constitute the first 15 elements of the array TAXIWAY, the branch taxiways going toward the gates the next 10, and the branch taxiways coming back from the gates the last 10. For clarity of understanding we use the constant nGates in the code instead of the number 10.

```
void configure_taxiways_and_runways(){
  rwCommon = new taxiway(0,0,0,0,0);  // common part of runways.

  TAXIWAYS[0] = new taxiway(RW1X1,RW1Y1,RW1X2,RW1Y2,tRW); // landing runway
  TAXIWAYS[1] = new taxiway(RW1X2,RW1Y2,TWX1,TWY1,tVT);   // right taxiway

  float twXdisp = ((float)TWX2-TWX1)/(nGates+1);
  float twYdisp = ((float)TWY2-TWY1)/(nGates+1);

  for(int i=0; i<= nGates; ++i){              // main taxiway: 11 segments
    TAXIWAYS[2+i] = new taxiway((int) (TWX1+i*twXdisp),
                                (int) (TWY1+i*twYdisp),
                                (int) (TWX1+(i+1)*twXdisp),
                                (int) (TWY1+(i+1)*twYdisp), tMT);
  }
  TAXIWAYS[3+nGates] = new taxiway(TWX2,TWY2,RW2X1,RW2Y1,tVT);  // left taxiway
  TAXIWAYS[4+nGates] = new taxiway(RW2X1,RW2Y1,RW2X2,RW2Y2,tRW);
                                                               // takeoff runway

  for(int i=0; i<nGates; ++i){                 // branch to gate
    TAXIWAYS[5+nGates+i] = new taxiway((int) (TWX1+(i+1)*twXdisp),
       (int) (TWY1+(i+1)*twYdisp),
       (int) (TWX1+(i+1)*twXdisp), TWYT, tBT);
  }
  for(int i=0; i< nGates; ++i){             // branch from gate
    TAXIWAYS[5+2*nGates+i] = new taxiway((int) (TWX1+(i+1)*twXdisp), TWYT,
 (int) (TWX1+(i+1)*twXdisp),
 (int) (TWY1+(i+1)*twYdisp), tBT);
  }
}
```

The names RW1X1 etc. are constants indicating the geometric coordinates of the appropriate taxiways, and the names tRW etc. are constants indicating the time to traverse the appropriate taxiways.

22.4 The plane class

The aircraft are implemented using a plane class. The aircraft are the entities in the simulation, and so plane must inherit from the simEntity class of the previous chapter and must provide a wakeup member function. In addition, an aircraft must appear on the screen as a part of the animation. So we inherit from the Turtle class as well. Indeed, our aircraft appear on the screen as turtles. We could have defined a more aircraft like visual appearance by using the polygon class, but that is left for the exercises.

The wakeup function of the plane class constitutes the heart of the simulation. Through successive executions of the function wakeup, the aircraft will move forward along the taxiways, request exclusive access to taxiways so that separation is maintained, make requests to allocate gates, wait at the gates and so on.

As discussed in Section ??, we must maintain some state in each plane object so that when wakeup is called we can decide what action to perform based on the state. What state do we need to maintain? You will realize that the action to be taken by the aircraft depends essentially upon its *position*. If the aircraft is in the middle of a taxiway, it merely needs to move forward whatever distance it can move in one unit time. When it comes to the end of a segment, it will need to first reserve the next segment (if any) and then turn to align with the direction of that segment. It will also need to release the segment on which it is currently located. Further, some special actions need to be taken if the aircraft is on specific segments, e.g. if the aircraft is on the segment before the main taxiway, then it must also ask for a gate to be allocated.

So clearly we need to keep track of which taxiway segment the aircraft is on. In addition, we must know how much the aircraft has travelled on the segment. We do this using 2 data members in each plane object: segment, and timeToSegmentEnd. In data member segment we store the index of the segment (in the vector TAXIWAY) on which the plane is currently present. On creation, this is set to -1, indicating that the plane is yet to enter the airport. In timeToSegmentEnd we store the number of steps we need to move forward before we need to worry about turning or reserving the next segment. In addition, we need to keep track of whether the aircraft has been allocated a gate, if so, which gate, and finally whether it has finished waiting for service, or is yet to begin the service. For the former we use an integer data member gate, and for the latter, a boolean, served. The former is initialized to a large value so that it indicates that a gate has not been allocated. The latter is initialized to false.

This leads to the following definition of plane.

```
class plane : public Turtle, public simEntity {
  int id;
  int arrivalT;
  int serviceT;
  int segment;
  int timeToSegmentEnd;
  int gate;
  bool served;
public:
  plane(int i, int at, int st) : id(i), arrivalT(at), serviceT(st) {
    timeToSegmentEnd = 0;
    segment = -1;   // currently before the landing runway.
    hide();
    penUp();
    gate = 10*nGates;  // very large number to indicate no gate allocated
    served = false;
  }
  void wakeup();
  void process_entry_to_next_segment();
  void enter(taxiway *ptw);
  bool getGate();
};
```

Note that some segment index values are special, e.g. index = 0 indicates the landing runway, and index = `TAXIWAYS.size()-1` the takeoff runway. To refer to such segments transparently we define the following names for later use.

```
const int toGateStart = 5+nGates, // starting index of taxiways to gates
      fromGateStart = 5+2*nGates; // starting index of taxiways from gates

enum segmentindices {preLanding = -1, landing = 0, requestGate = 1,
     preFirstGate = 2, preTakeOff = toGateStart-2,
     takeOff = toGateStart-1};
```

We will of course not use `PRELANDING = -1` to index `TAXIWAYS`, but this can effectively be used to express that the aircraft is yet to arrive into the airport.

22.4.1 The function `wakeup`

At a very high level, the `wakeup` member function is simple. If `timeToSegmentEnd` is not zero, we generally only move forward in the current segment. If `timeToSegmentEnd` has become 0, we are entering a new segment, and might need to take some decisions.

```
void plane::wakeup(){
  if(timeToSegmentEnd == 0) process_entry_to_next_segment();
  else{
    if((segment == landing || segment == takeOff)
       && timeToSegmentEnd == TAXIWAYS.at(segment)->traversalT/2){
      rwCommon->release();
    }
    forward(TAXIWAYS[segment]->stepsize);
    --timeToSegmentEnd;
    simQueue::insert(1,this);
  }
  return;
}
```

If timeToSegmentEnd is not 0, we move forward by the stepsize determined for the current taxiway, unless we are on the landing segment or the takeoff segment. Remember that as we pass the middle of these segments, we must release rwCommon. This is what the above code does. At the end of the move, the plane puts itself back in simQueue to be woken up again at the next step.

The function process_entry_to_next_segment is given next. The code in this consists of a long sequence of cases depending on the state of the plane. In each case, a certain set of actions are performed. After those actions are performed, the plane may either have to wait because certain resources are not available, or if the resources may immediately go to the next state. The plane may be in a position to immediately execute the actions of the next state, and so it will have to reexecute the code for wakeup, the easiest way to do this is by queueing itself in simQueue for the current step itself. Here is the first part of the function.

```
void plane::process_entry_to_next_segment(){
  if(segment == preLanding){
    if(!TAXIWAYS[0]->reserve(this)) TAXIWAYS[0]->await(this);
    else if(!rwCommon->reserve(this)) rwCommon->await(this);
    else{
      simQueue::log()<< "Plane " << id << " lands. scheduled arrival "
     << arrivalT << ", Service time " << serviceT << endl;
      segment = 0;
      show();
      enter(TAXIWAYS[0]);
    }
  }
// to be continued
```

This part describes what is to be done for the first time wakeup is called, i.e. when the plane is yet to land. The plane must acquire the landing runway, i.e. TAXIWAY[0]. If this is not immediately available, then the plane waits for it by calling the await function. Else it proceeds to acquiring rwCommon. If both these taxiways are available, then it prints out a status message. The member segment is set to 0, indicating it has landed. And finally the member function enter is called to do some bookkeeping associated with entry to a segment. This function aligns the plane with the line associated with the segment, and sets timeToSegmentEnd to be the time required to traverse this segment. Finally, the plane is put back on simQueue for executing at the current timestep itself (Section 22.4.2).

An important point should be noted. If any of the resources needed, e.g. rwCommon is not available, the plane awaits its release. It might be tempting to think that the execution of process_entry_to_next_segment "suspends" at the point of calling await and resumes when the resource becomes available. However, the actual execution is more complex: when the resource is released, the wakeup function is first called. The function executes from the beginning and trace the same path as before, into process_entry_to_next_segment, except that this time the resource will be seen to be available, i.e. say rwCommon->reserve(this) will turn out true, and hence the else part will be entered.

The next call to wakeup happens when the plane arrives at the end of segment 0. In this case, we simply need to reserve the next segment and so on, no special actions are needed.

It is convenient to organize the code of process_entry_to_next_segment so that the special cases come up first. So the next special case concerns entry to segment 1, where the plane must make a request for a gate.

```
// process_entry_to_next_segment continued
  else if(segment == requestGate){
    if(!getGate()) simQueue::insert(1,this);
    else if(!TAXIWAYS.at(segment+1)->reserve(this))
      TAXIWAYS.at(segment+1)->await(this);
    else{
      TAXIWAYS.at(segment)->release();
      segment++;
      enter(TAXIWAYS.at(segment));
    }
  }
```

Here we check to see if a gate is available, and if not, we retry after 1 step. We cannot simply wait on a resource, because we are waiting for *any* of the gates to be available, as will be seen in the definition of getGate (Section 22.4.3). Hence we need to actively try again. The Exercises ask you to explore how to avoid this repeated checking.

If some gate i is available, the function getGate sets the member gate, to i. After that the plane continues taxiing and tries to move to the next segment. If that segment is available, the plane releases the current segment and enters it. Note that the release must happen only after the next segment has been acquired.

The next cases are about turning towards the gate, waiting, and returning back to the main taxiway.

```
// process_entry_to_next_segment continued
  else if(segment == preFirstGate + gate){  // about to turn to gate?
    TAXIWAYS[segment]->release();
    segment = toGateStart + gate;
    enter(TAXIWAYS[segment]);
  }
  else if(segment == toGateStart + gate){  // at end of taxiway to gate?
    if(!served){
      simQueue::log()<< " Plane " << id << " at gate " << gate
     << " will wait for " << serviceT << endl;
      served = true;
      simQueue::insert(serviceT,this); // wait for service
    }
    else{
      segment = fromGateStart + gate;
      enter(TAXIWAYS[segment]);
    }
  }
  else if(segment == fromGateStart + gate){   // at end of from taxiway?
```

```
    if(!TAXIWAYS[preFirstGate + gate + 1]->reserve(this))
      TAXIWAYS[preFirstGate + gate + 1]->await(this);
    else{
      TAXIWAYS[toGateStart + gate]->release();
      segment = preFirstGate + gate + 1;
      enter(TAXIWAYS[segment]);
    }
  }
```

The condition check `segment == preFirstGate + gate` will succeed if the current segment is the one at which we need to turn towards our assigned gate, `gate`. Note that on initialization, we set `gate` to a large value, so that the condition check would have no chance of succeeding until we gave a valid value in `gate`. If the check succeeds, we move to a branch taxiway. This is easily seen to correspond to the segment `toGateStart + gate`. So we enter that.

The next case concerns the situation when wakeup is called with `segment == toGateStart + gate`. Clearly, this is when we are at the end of the branch segment. After we have just turned into the branch taxiway, the data member `served` would be false. So we set wait to simulate the service time and set it `true`. If `served` was already `true`, we start our journey towards the main taxiway by entering the branch taxiway going away from the gate `gate`. This corresponds to segment `fromGateStart + gate`.

The final case is when we have reached the end of the branch taxiway going towards the main taxiway, i.e. segment `fromGateStart + gate`. In this case we must enter the main taxiway, after releasing the reservation on the taxiway going towards our allocated gate. As we noted, this will enable other planes to use this gate later.

The final two special cases concern the takeoff and pre-takeoff segments.

```
// process_entry_to_next_segment continued
  else if(segment == preTakeOff){
    if(!TAXIWAYS[takeOff]->reserve(this)) TAXIWAYS[takeOff]->await(this);
    else if(!rwCommon->reserve(this)) rwCommon->await(this);
    else{
      TAXIWAYS[segment]->release();
      ++segment;
      enter(TAXIWAYS[segment]);
    }
  }
  else if(segment == takeOff){
    TAXIWAYS[segment]->release();
    hide();
    simQueue::log() << " Plane " << id << " left." << endl;
  }
```

When leaving the pre-takeoff segment, we must not only reserve the takeoff segment but also `rwCommon`. And when we leave the takeoff segment, we must print out a message and hide ourselves.

Finally, the default case, which applies to all non-special segments.

```
// process_entry_to_next_segment continued
  else{  // ordinary segment
    if(!TAXIWAYS[segment+1]->reserve(this))
      TAXIWAYS[segment+1]->await(this);
    else{
      TAXIWAYS[segment]->release();
      ++segment;
      enter(TAXIWAYS[segment]);
    }
  }
}   // end of function
```

We reserve the next segment and enter it, after releasing the current segment.

22.4.2 The function `enter`

```
void plane::enter(taxiway* ptw){
  Position linestart = ptw->getStart();
  moveTo(linestart.getX(), linestart.getY());
  Position lineend = ptw->getEnd();
  face(lineend.getX(), lineend.getY());
  timeToSegmentEnd = ptw->traversalT;
  simQueue::insert(0,this);
}
```

When entering a segment, the plane first positions itself at the beginning of the line corresponding to the segment. Most of the time the end of the last segment and the beginning of the next segment will be identical, so this step is really not needed. However, when the plane lands, the positioning is required. After that the plane orients itself by facing the end of the line representing the taxiway. Then the counter `timeToSegmentEnd` is initialized to the time required to traverse this segment. Now the plane is ready to travel on the segment, and for this it inserts into the queue to be woken up immediately.

22.4.3 The function `getGate`

```
bool plane::getGate(){
  for(int i=0;i<nGates;++i)
    if (TAXIWAYS[toGateStart + i]->reserve(this)){
      gate = i;
      gateAllocated = true;
      return true;
    }
  return false;
}
```

Remember that the taxiway going towards gate `i` is to be reserved to simulate acquisition of gate `i`. This taxiway is represented by segment `toGateStart + i`.

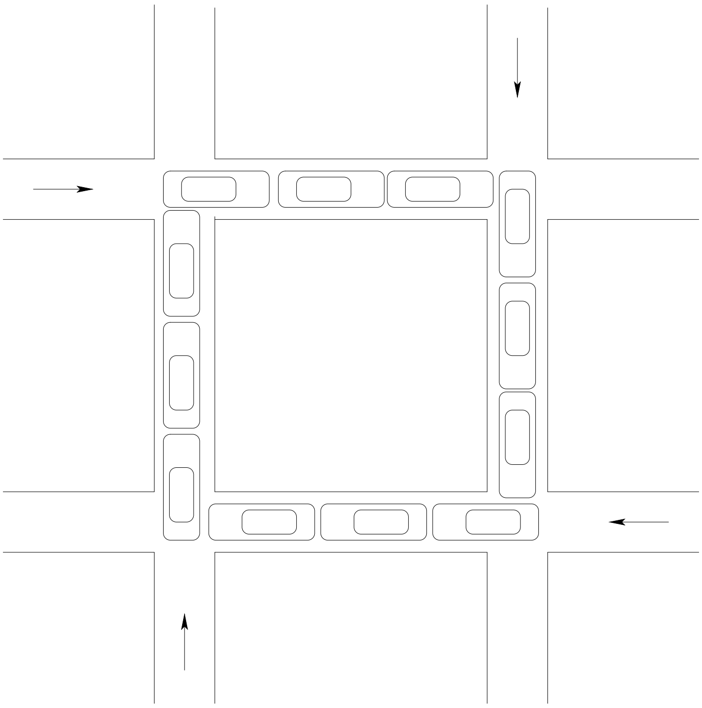


Figure 22.2: Traffic deadlock in a city

22.5 Deadlocks

A *deadlock* is a technical term used to describe a system in which one entity e_1 is waiting to reserve a resource held by entity e_2 which in turn is waiting to reserve a resource held by and entity e_3 and so on, till some entity e_n in this sequence is waiting to reserve a resource held by e_1. Notice that in this case no entity can make progress, because all are waiting for each other. Figure 22.5 shows cars deadlocked on roads in a city. Note that the roads are one ways, as shown. The cars are waiting for the space ahead of them to become empty, but it never will, as you can see.

A deadlock is possible on a circular taxiway if every segment contains a plane which wants to move forward. In our airport it would seem that the taxiways do not form a circular path. However we have to be careful in implementing the half-runway-exclusion rule.

It turns out that deadlocks will not arise if we observe the following discipline in reserving `rwCommon`. A landing aircraft must first reserve the landing runway and only then `rwCommon`. Similarly a plane taking off must first reserve the takeoff runway and only then `rwCommon`. You can see that this is a good strategy: `rwCommon` being a precious resource must be reserved last. If a plane reserves `rwCommon` and cannot reserve the landing runway, then it prevents take offs unnecessarily until such time as it reserves the landing runway. More formally, as the exercise asks you, you should be able to prove that if this policy is used there can be no deadlock. On the other hand, if landing planes as well as planes taking off reserve `rwCommon` first, then it is possible to create a deadlock by carefully constructing the arrival sequence of the planes. The exercises invite you to explore this possibility.

22.6 On Global variables vs. reference variables

This is the only chapter in which we have used a *global variable*. As discussed in Appendix B, use of global variables is not recommended.

Instead of making `TAXIWAYS` a global variable, we should really pass a reference to it to the `plane` class in which it is used.

Exercises

1. Modify the simulation program to print out the average aircraft delay.

2. Define a better `plane` class in which the on screen image looks like an aircraft rather than a triangle.

3. Suppose we wish to ensure that as much as possible, an aircraft must land at its arrival time. Thus, while granting `rwCommon` to a departing plane, we must check whether no plane will want to land during the interval in which the departing plane will use `rwCommon`. Device a good mechanism to do this. Hint: perhaps you can reserve the landing runway and `rwCommon` a bit earlier than needed?

4. The program given in the text uses so called *busy waiting* to allocate gates, i.e. if a gate is not currently available, the plane retries after 1 step. It will be more efficient if the plane can `await` the release of *any* gate. Develop a class to represent such a resource group. A resource group models a sequence of objects, each of which can be either reserved or unreserved. On a `reserve` request, one of the unreserved objects must be allocated, i.e. the requesting entity should be set as its owner. If all objects are currenly reserved, then the reserve request is deemed to fail and should thus return `false`. In that case the entity may `await` its release. When any of the objects becomes available, that should get reserved for the waiting entity. Use this in the simulation code.

5. Show that our strategy of reserving resources ensures that there is no deadlock. Specifically, show that at every step some aircraft will make progress, and that there will not exist entities $e_0, \ldots, e_{n-1}$ where e_i is waiting for a resource currently held by entity $e_{i+1 \bmod n}$.

6. Suppose we reserve `rwCommon` first and then the take off or landing runways. Construct an input sequence (the file `arrivals.txt`) such that there is a deadlock.

7. Consider the shortest path algorithm of Section 21.4. Suppose that we are also given the geometric coordinates for each vertex of the graph. Show a visual simulation of the algorithm, i.e. a turtle should move along each edge as if it were a cyclist.

8. Suppose we do not want to divide the taxiway into segments and require that there is at most one aircraft in each segment. Instead, suppose we will allow a plane to move a certain stepsize at each step while keeping a certain safe distance behind the plane ahead, if any. Implement this. The other rules must still be followed, i.e. the

half-runway-exclusion rule and the rule that there can be at most one aircraft on each runway at any instant. Also, there can be only one aircraft on any branch taxiway.

9. Simulate another airport configuration of your choosing.

Chapter 23

Non-linear simultaneous equations

Suppose you want to construct a parallelopiped box of volume 1010 cm^3, surface area 700 cm^2 and having a base whose diagonal is 22 cm. What are the lengths of the sides of the box? If u_1, u_2, u_3 denote the side lengths in cm, clearly we have $u_1u_2u_3 = 1010$, $2(u_1u_2 + u_2u_3 + u_3u_1) = 700$ and $u_1^2 + u_2^2 = 22^2$. We have 3 equations in 3 unknowns, but unfortunately these equations are non-linear! In Section **??** we saw how to solve linear simultaneous equations, but this problem is much more difficult. Indeed it is fair to say that solving non-linear simultaneous equations is bit of an art. While, there is no single guaranteed method for solving non-linear equations in many variables, there are some strategies which seem to often work. Once such strategy is the Newton-Raphson method (NRM). We have already studied NRM in Section 7.3 for the one dimensional case. Its generalization to multiple dimensions is precisely what we need and we will study it in this chapter.

After studying NRM for multiple dimensions, we will consider a more elaborate problem: given a chain of links of different lengths, compute the configuration in which it hangs if suspended from some fixed pegs. We will see that NRM solves this problem nicely.

The exercises give more applications of NRM in multiple dimensions.

23.1 Newton-Raphson method in many dimensions

In one dimension, NRM is used to find the root of a function f of one variable, i.e. find u such that $f(u) = 0$. The higher dimensional case is a natural generalization. We are now given n functions $f_1, \ldots, f_n$ each of n variables, and we want to find their *common* root, i.e. a set values $u_1, \ldots, u_n$ such that $f_i(u_1, \ldots, u_n) = 0$ for all i.

As you might see, this is really the same as solving simultaneous, possibly non-linear equations. Any equation in n unknowns can be written so that the right hand side is 0, but then we can treat what is on the left hand side as a function of the unknowns. Indeed, our equations for the box problem can be stated in this form as follows:

$$
\begin{aligned}
f_1(u_1, u_2, u_3) &= u_1u_2u_3 - 1010 = 0 && (23.1)\\
f_2(u_1, u_2, u_3) &= 2(u_1u_2 + u_2u_3 + u_3u_1) - 700 = 0 && (23.2)\\
f_3(u_1, u_2, u_3) &= u_1^2 + u_2^2 - 484 = 0 && (23.3)
\end{aligned}
$$

Indeed the common root $u = (u_1, u_2, u_3)$, of f_1, f_2, f_3 will precisely give us the side lengths of the box we want to construct.

An important point to be noted is that each function f_i can be thought of as the error for the corresponding equation. Our goal in solving the equations is to make the error zero. Note that in this interpretation the errors can be positive or negative.

As in one dimension, NRM in many dimensions proceeds iteratively. In each iteration, we have our current guess for the values of the unknowns. We then try to find by how much each unknown should change, so that we (hopefully) get closer to the root. We then make the required change in the unknowns, and that becomes our next guess. We check if we our new values are close enough to the (common) root. If so, we stop. Otherwise we repeat. We will use $u_1, \ldots, u_n$ to denote the unknowns. Let their current values be $u_{1cur}, \ldots, u_{ncur}$. Our goal is to determine what increments $\Delta u_1, \ldots, \Delta u_n$ to add to these values so as to get our next guess $u_{1next}, \ldots, u_{nnext}$.

We will use our box problem as a running example for explaining the method. Suppose for this problem we have guessed $u_{1cur} = 20$, $u_{2cur} = 10$ and $u_{3cur} = 5$. Then we have $f_1(u_{1cur}, u_{2cur}, u_{3cur}) = -10$, $f_2(u_{1cur}, u_{2cur}, u_{3cur}) = -0$, $f_3(u_{1cur}, u_{2cur}, u_{3cur}) = 16$,

For a minute consider that we only want to make f_1 become 0, and we only can change u_1. Now this is simply the one dimensional case. If we make a small increment Δu_1 in u_1, we know that f_1 will change in proportion to Δu_1 and the derivative of f_1 with respect to u_1. Actually, since f_1 is a function of many variables, all of which we are keeping fixed except for u_1, we should really say, "the partial derivative of f_1 with respect to u_1". Thus the (additive) change Δf_1 in f_1 will be approximately $\frac{\partial f_1}{\partial u_1}\Delta u_1$. Further, the partial derivative must be evaluated at the current values, so we will write:

$$\Delta f_1 \approx \left.\frac{\partial f_1}{\partial u_1}\right|_{cur} \Delta u_1 \tag{23.4}$$

But we want $f_{1next} = f_{1cur} + \Delta f_1$ to become zero, so perhaps we should choose $\Delta f_1 = -f_{1cur} = 10$. Further, we know that $\left.\frac{\partial f_1}{\partial u_1}\right|_{cur} = u_2 u_3\Big|_{cur} = 50$. Thus we have:

$$10 \approx 50\Delta u_1$$

So we indeed choose $\Delta u_1 = \frac{10}{50} = 0.2$. The new value of u_1 becomes 20+0.2=20.2, and the new value of f_1 becomes $20.2 \times 10 \times 5 - 1010 = 0$. In this case, the error in the first equation has completely vanished. Things will not be this good in general, but as you might remember from Section 7.3 that we can expect f_1 to get closer to 0 than it was.

But of course, nothing really forces us to only change u_1. Thus, if we are allowed to vary all the variables, then equation 23.4 generalizes as:[1]

[1] Think about changing u_1 first, and then u_2 and so on. Initially we are at $(u_{1cur}, u_{2cur}, u_{3cur})$. After changing u_1 by get to the point which we will call $(u_{1cur'}, u_{2cur'}, u_{3cur'})$. In this movement, we have changed f_1 by about $\left.\frac{\partial f_1}{\partial u_1}\right|_{cur} \Delta u_1$. From the new point we change u_2 by Δu_2. The change that this causes in f_1 is about $\left.\frac{\partial f_1}{\partial u_2}\right|_{cur'} \Delta u_2$ total change in f_1 is approximately

$$\left.\frac{\partial f_1}{\partial u_1}\right|_{cur} \Delta u_1 + \left.\frac{\partial f_1}{\partial u_2}\right|_{cur'} \Delta u_2$$

But the values at *cur* and *cur*′ are nearly the same, assuming $\Delta u_1, \Delta u_2$ are small.

$$\Delta f_1 \approx \left.\frac{\partial f_1}{\partial u_1}\right|_{cur} \Delta u_1 \;+\; \left.\frac{\partial f_1}{\partial u_2}\right|_{cur} \Delta u_2 \;+\; \left.\frac{\partial f_1}{\partial u_3}\right|_{cur} \Delta u_3 \tag{23.5}$$

Again, we want $\Delta f_1 = -f_{1cur}$, and try to pick $\Delta u_1, \Delta u_2, \Delta u_3$ to satisfy

$$-f_{1cur} = \left.\frac{\partial f_1}{\partial u_1}\right|_{cur} \Delta u_1 + \left.\frac{\partial f_1}{\partial u_2}\right|_{cur} \Delta u_2 + \left.\frac{\partial f_1}{\partial u_3}\right|_{cur} \Delta u_3 \tag{23.6}$$

In a similar manner we will require the following as well.

$$-f_{2cur} = \left.\frac{\partial f_2}{\partial u_1}\right|_{cur} \Delta u_1 + \left.\frac{\partial f_2}{\partial u_2}\right|_{cur} \Delta u_2 + \left.\frac{\partial f_2}{\partial u_3}\right|_{cur} \Delta u_3 \tag{23.7}$$

and

$$-f_{3cur} = \left.\frac{\partial f_3}{\partial u_1}\right|_{cur} \Delta u_1 + \left.\frac{\partial f_3}{\partial u_2}\right|_{cur} \Delta u_2 + \left.\frac{\partial f_3}{\partial u_3}\right|_{cur} \Delta u_3 \tag{23.8}$$

Notice now that $\Delta u_1, \Delta u_2, \Delta u_3$ are the only unknowns in Equations (23.6,23.7,23.8), and in these variables the equations are linear! Thus we can solve them. Evaluating the current values of the functions and the partial derivatives we get:

$$\begin{aligned} 10 &= \quad 50\Delta u_1 + 100\Delta u_2 + 200\Delta u_3 && (23.9)\\ 0 &= \quad 30\Delta u_1 + 50\Delta u_2 + 60\Delta u_3 && (23.10)\\ 16 &= \quad 40\Delta u_1 + 20\Delta u_2 && (23.11) \end{aligned}$$

Solving this we get $(\Delta u_1, \Delta u_2, \Delta u_3) = (-0.52, 0.24, 0.06)$. So we have $(u_{1next}, u_{2next}, u_{3next} = (19.48, 10.24, 5.06)$. For these values we see that $(f_1, f_2, f_3) = (0.655554, 0.283244, -0.327963)$, which taken together are much closer to zero than $(10, 0, 16)$.

23.1.1 The general case

In general we will have n equations:

$$-f_i(u_{1cur}, \ldots, u_{ncur}) = \sum_j \left.\frac{\partial f_i}{\partial u_j}\right|_{cur} \Delta u_j \tag{23.12}$$

We solve these to get $(\Delta u_1, \ldots, \Delta_n)$, and from these we can calculate $u_{jnext} = u_{jcur} + \Delta u_j$, for all j.

It is customary to consider the above equations in matrix form. Define an $n \times n$ matrix A in which $a_{ij} = \left.\frac{\partial f_i}{\partial u_j}\right|_{cur} \Delta u_j$. Define an n element vector b where $b_i = -f_i(u_{1cur}, \ldots, u_{ncur})$. Let Δu denote the vector of unknowns $(\Delta u_1, \ldots, \Delta u_n)$. Then the above expressions can be written in the form

$$A(\Delta u) = b$$

in which A, b are known and we solve for Δu. The matrix A is said to be the Jacobian matrix for the problem. Further, it is customary to define vectors $u_{cur} = (u_{1cur}, \ldots, u_{ncur})$ and $u_{next} = (u_{1next}, \ldots, u_{nnext})$. Then our next guess computation is simply:

$$u_{next} = u_{cur} + \Delta u$$

Next we comment on when we should terminate the procedure, and how to make the first guess.

23.1.2 Termination

We should terminate the algorithm when all f_i are close to zero. A standard way of doing this is to require that $\sqrt{f_1^2 + \cdots, f_n^2}$ become smaller than some ϵ that we choose, say $\epsilon = 10^{-7}$ if we use `float`, and even smaller if we use `double` to represent our numbers. In keeping with our interpretation that f_i is the error, the quantity $(f_1, \ldots, f_n)$ is the vector error, and $\sqrt{f_1^2 + \cdots, f_n^2}$ is the 2-norm or the Euclidean length of the vector error.

23.1.3 Initial guess

Finding a good guess to start off the algorithm turns out to be tricky. In one dimension, we roughly plotted the function and sought a point close enough to the root. In multiple dimensions, this is more difficult.

Newton's method works beautifully if we are already close to the root. This is because very close to the root, the equations such as Equation (23.6) become very accurate. One idea is to try to satisfy the equations approximately. It is often enough to satisfy only some of the equations. For example, we found for the box problem that a starting guess of $(u_1, u_2, u_3) = (9, 10, 11)$ work quite well. Note that these numbers satisfy Equation 23.1 very closely.

On the other hand, an initial guess of (1,2,3) worked quite badly: it produced the "answer" (2.2659 -21.883 -20.3692). Note that this satisfies the equations closely, but surely we cannot have negative side lengths! This points to another feature of non-linear equations: there may be multiple roots. Your iterative procedure may not necessarily take you to the correct one.

In the next section, we will have more to say on this.

23.2 How a necklace reposes

Suppose you are given a chain of n links, where the ith link has length L_i, $i = 0, \ldots, n-1$. Say the chain is hung from pegs at points (x_0, y_0) and (x_n, y_n) which are known. What is the shape attained by the chain when it comes to rest, hung in this manner? The links in the chain need not have equal lengths.

23.2.1 Formulation

Let x_i, y_i denote the coordinates of the left endpoint of link i, and of course x_{i+1}, y_{i+1} are then the coordinates of the right endpoint, where $i = 0, \ldots, n-1$. As discussed above, we already know the values of (x_0, y_0) and (x_n, y_n), these are the coordinates of the pegs from which the chain is suspended. The other x_i, y_i are the unknows we must solve for. We are given the lengths L_i of the links, thus the variables x_i, y_i must satisfy the following equations.

$$(x_{i+1} - x_i)^2 + (y_{i+1} - y_i)^2 - L_i^2 = 0 \tag{23.13}$$

We also need to consider the forces on the links. Suppose that F_i is the (vector) force exerted by link $i-1$ on link i (F_0 is the force exerted on link 0 by the left peg, and F_{n-1} is the force exerted by link $n-1$ on the right peg). Note of course that by Newton's third law, if one object exerts a force F on another, the latter exerts a force of $-F$ on the former. So each link i has a force F_i acting on its left endpoint, and a force $-F_{i+1}$ acting on its right endpoint. Further, there is its weight, W_i, also vector, which acts at its center. When the chain is at rest, total force on each link must be zero, as well as the total torque.

Thus for each link we have $F_i - F_{i+1} + W_i = 0$. Suppose $F_i = (h_i, v_i)$, i.e. h_i and v_i are the horizontal and vertical components of F_i. Because W_i is only vertical, we can write it as $W_i = (0, w_i)$. The weight acts downwards, so perhaps we should write $-w_i$ as the y component. However, do note that in the coordinate system of our graphics screen y increases downwards. Hence we do not have the negative sign. Further, we will assume that the weight is proportional to the length, and so we will write $w_i = L_i$. Now balancing the horizontal component we get $h_i - h_{i+1} = 0$, i.e. all these variables are identical! Thus we could write a common variable h instead of them. Balancing the vertical component we get, for all i:

$$v_i - v_{i+1} + L_i = 0 \tag{23.14}$$

Finally we need to balance the torque as well. For this we need to consider the right endpoint to be the center. Remember that the torque due to a force F equals the magnitude of the force times the perpendicular distance from the center to the line of the force. The torque due to the horizontal component of F_i is simply the horizontal component times the vertical distance to the horizontal component. Thus it is $h_i(y_{i+1} - y_i) = h(y_{i+1} - y_i)$. This torque is in the clockwise direction. The torque due to the vertical component is similar, $v_i(x_{i+1} - x_i)$, but in the counter clockwise direction. The distance to the line of the weight is $(x_{i+1} - x_i)/2$, and so the torque due to it is $L_i(x_{i+1} - x_i)/2$, also in the counterclockwise direction. But the total torque, considered in say the clockwise direction, must be zero. Thus we get:

$$h(y_{i+1} - y_i) - v_i(x_{i+1} - x_i) - L_i(x_{i+1} - x_i)/2 = 0 \tag{23.15}$$

Equations (23.13), (23.14), and (23.15) apply to each link, and thus we have $3n$ equations over all. The unknowns are $x_1, \ldots, x_{n-1}$ (noting that x_0, x_n are known) and similarly $y_1, \ldots, y_{n-1}$, and h, and $v_0, \ldots, v_n$. Thus there are a total of $(n-1)+(n-1)+1+(n+1) = 3n$ unknowns. Thus the number of unknowns and the number of equations match; however, our equations (23.13) and (23.15) are not linear. So we need to use the Newton Raphson method.

23.2.2 Initial guess

Making a good initial guess is vital for this problem.

To make a good guess, we have to make use of our "common sense" expectation about what the solution is likely to look like. For the necklace problem, we can expect that the necklace to hang in the shape of a "U". So presumably we can set (x_i, y_i) along a semicircle which hangs from the pegs. Also we can arrange the force values so that the total vertical force on each link is 0. One way to do this is to compute the total weight, and set v_0, v_n to bear half of it. Once we set this the other values of v_i can be set as per Equations (23.14). The horizontal force h could be set to 0 to begin with. It is much trickier to try to balance the torque. But it turns out that the initial values as we have outlined here are enough to produce a good answer.

23.2.3 Experience

We coded up the algorithm and set the initial values as per the guessing strategy described above. We then ran the algorithm. After each iteration, we plotted the necklace configuration on our graphics screen. As you can see, the configuration quickly seems to reach a stable point. Indeed, we also printed out the square error, and it got close to zero fairly quickly.

23.3 Remarks

As you experiment with NRM you might notice that the error norm (as defined in Section 23.1.1) does not necessarily decrease in each iteration. This is understandable, the error norm is guaranteed to decrease only if the equations such as (23.6) hold exactly.

It is possible to show, however, that the vector Δu does indeed give the exact direction in which to move from u_{cur} for which the rate of reduction of the error norm is the largest possible. Thus there exists an α such that the error norm at $u_{cur} + \alpha\Delta u$ will be strictly smaller than that at u_{cur}. We can try to roughly find this α by starting with $\alpha = 1$ (which is equivalent to taking the full step, ie. the basic NRM), and successively halving it till we find a point $u_{cur} + \alpha\Delta u$ where the error norm is lower than at u_{cur}.

23.4 Exercises

1. Write the program to solve the box problem.

2. Write the program to solve the chain link problem. Display the configuration of the chain on the screen after each iteration of NRM.

3. Implement the idea of finding an α using which we ensure that the error decreases in every iteration. For the chain link problem, you will see that this gives smoother movement towards the final solution.

4. Something on chemical equations?

Appendix A

Installing Simplecpp

It should be possible to install simplecpp on any system which has X Windows (X11) installed. We have installed simplecpp on Ubuntu, Mac OS X, and Microsoft Windows running Cygwin/X. To install, download

```
www.cse.iitb.ac.in/~ranade/simplecpp.tar
```

untar it, and follow the instructions in simplecpp/README.txt.

You will need to have the GNU C++ compiler, which is present on all the systems mentioned above.

Appendix B

Scope and Global Variables

When you declare a name in a program, the name is accessible only for a part of the program. This region of the program where the name is accessible is said to be its *scope*. In this chapter we discuss the general rule for determining the scope of a name.

We also discuss the notion of a global variable.

We begin by clarifying the notion of a *block*.

B.1 Blocks

So far, we have defined a block to be a group of statements which are separated by semicolons, and grouped together between a left brace { and a right brace }.

For the purpose of the following discussion, the notion of a block must be modified slightly in the case of functions and `for` statements. It is customary to consider the entire definition of a function to be a block (even though the braces "{" and "}" only enclose the body of the function). Likewise, the entire `for` statement is considered to be a block, and not just the body of the for statement.

There is one more way in which the notion of a block must be modified: the entire file is also to be thought as constituting a block. We will call this the *global* block.

B.2 Scope

First consider the case in which a name is declared only once in the entire program. In this case the rule is simple: the name is accessible in the region of the program text starting at the declaration and going to the end of the innermost block containing the declaration. This region is the scope of the variable.

Note that by our extended notion of a block as discussed above, the parameters of the function are accessible throughout the entire body of the function. Also if a new variable is declared in the `initilization` part of the `for`, it is accessible inside the entire `for` statement.

The case in which a variable is declared several times is more complicated. First, it is important to note that multiple declarations of the same name are not always allowed. Define the *parent* block of a declaration to be the innermost block containing the declaration.

Then multiple declarations are allowed only so long as each declaration has a distinct parent block. As an example, consider the declarations of k in the following code fragment.

```
int k=10;                 // first declaration
k = k - 1;                // first arithmetic statement
cout << k << endl;        // first print statement
{                         // block B begins
  cout << k << endl;      // second print statement
  int k=11;               // second declaration
  cout << k << endl;      // third print statement
}                         // block B ends
int k=12;                 // third declaration, NOT ALLOWED.  ERROR
{                         // block C begins
  cout << k << endl;      // fourth print statement
}                         // block C ends
```

The parent of the first and third declarations is the global block, so this code is in error. So let us assume that the third declaration is not present. Now there are only two declarations, the parent of the first is the global block, and that of the second is the block named B in the code. Since the parent blocks are different, this code (without the third declaration) is acceptable. Note that in this code two distinct variables of the same name k will be created, in the usual manner, when the corresponding statement is encountered during execution. We will need a rule to determine the scope of each of these declarations (or the associated variables).

The rule for this is a slight modification to the simple rule stated earlier. Let $d_1, d_2, \ldots, d_n$ be declarations of the same name k, in order from top to bottom. We then first find the scopes $s_1, s_2, \ldots, s_n$ of each of these declarations as per the simple rule. Note that each s_i is just a region of the code. Because of the distinct parent rule above, it will turn out for $i < j$ that s_i, s_j will either be disjoint, or s_i will contain s_j. If some s_i contains s_j, then we will say that d_i is being *shadowed* by d_j in the region s_j. We can now define the actual scope of a declaration d_i: it is the region s_i, from which we subtract the regions in which d_i is shadowed by another declaration.

The scope can also be defined differently as follows. Consider a statement S in the code. It will be said to be in the scope of a declaration d_i if i is largest such that the simple scope s_i defined above contains S.

Going back to our example, our first and second declarations now become d_1, d_2. The scope of d_1 is the entire code fragment, except for the third print statement; in the third print statement d_2 shadows d_1. The scope of d_2 is just the third print statement.

Now we know how the code fragment will execute. Based on the discussion above, we can associate a declaration with every occurrence of a name, i.e. the declaration in whose scope that occurrence is. Then we can determine what value the name represents, or what is to be updated. Thus the first arithmetic statement is in the scope of the first declaration, so the variable associated with this declaration will be used. This will cause the value of the variable to change to 9 from 10. The first print statement is also in the scope of the same declaration. Thus this will cause the value 9 to be printed. The second print statement is also in the scope of the first declaration, so this will again cause the value 9 to be printed.

The third print statement, however, is in the scope of the second declaration, and hence the value 11 will be printed. Remember that the third declaration is illegal, and we decided that we will remove it from the code. The fourth print statement is in the scope of the first declaration, so the value 9 will be printed again.

I hope you dont find these rules too confusing. Indeed, the scope rules are expected to be convenient. If you want a new variable inside a block, you just go ahead and declare it without worrying whether it interferes with some variable declared outside. Just in case the same name was used earlier, your declaration will shadow the variable within the block and not interfere with it outside the block.

B.3 Creation and destruction of variables

In C++ it helps to be clear about when variables are created and when they get destroyed.

A variable is created when the statement which defines it is executed. It gets destroyed when control finishes executing the last statement in the innermost block containing the definition and exits this block. Note that control may leave this block temporarily, say for executing a function call. In this case nothing happens to the variables. They will be available for use when control returns from the called function.

At the time of the creation of the variable, the constructor for the variable is called. At the time of destruction, the destructor for the variable gets called.

The rule is slightly tricky for variables declared in loops. If a variable is declared in the body of the loop, then it will get destroyed after each iteration, and be created again when the control enters the loop body for another iteration. Similar is the case for functions: the local variables defined in a function get created when control reaches the corresponding statement and destroyed when control leaves the function. If you want the variables to retain their values from one iteration to the next, or from one call to the other, then you can declare them to be `static`, i.e. prefix this keyword in the definition. If the definition includes initialization, then the initialization will happen only on the first call or only on the first loop iteration.

Variables defined in the `initialization` part of a `for` statement (Section 6.6) however behave slightly differently. These variables are created before the first iteration executes, and destroyed only after the last iteration executes.

The notion of `static` members in classes should not be confused with the use of the keyword here.

B.4 Global variables

It is possible to declare variables outside of all functions (including the main program), in the global block. In that case, the variable thus declared will be visible in all the functions (unless it is shadowed), provided the declaration appears above the respective function definition. If there are functions, and the function definitions appear below the declaration of the variable, then the variable is visible in the functions as well (unless it is shadowed).

Variables declared outside the main program are called *global variables*.

Global variables can serve as a means of exchanging information between functions. Indeed, the main program (or any function) could write data into a global variable, and then call another function, which can read the global variable and get the data. Similarly, instead of returning a value, the function could simply write the value into a global variable, which the main program could then read. This way of using global variables may appear convenient, but indeed it is fraught with danger. This is because global variables make it harder to reason about code.

Suppose you see a statement:

```
pqr = f(abc,def);
```

If we know that `f` does not access global variables in anyway (and such functions are often called *pure* functions) then we know that the `f` can affect only the variable `pqr`, and further, that effect only depends upon the variables `abc` and `def`. If `f` does refer to global variables, then we need to look at the code to decide what really affects the execution of `f`. It is actually even more complicated – having noted that `f` refers to a global variable `ghi`, we then need to search the code to see which is the last statement to set `ghi`. All this is avoided for pure functions. So the general consensus is: avoid the use of global variables as much as possible.

However, there may be some variables in your program that may somehow be very important and hence needed in most functions. In such cases it is acceptable to make these variables global; passing them as arguments to every function might make the code too verbose.

Appendix C

Managing Heap Memory

In Section 16.1 we discussed heap memory. We mentioned that managing heap memory is tricky. The simplest way to use heap memory is to use STL classes such as vectors, maps, queues, strings. These objects hide the memory management from the user, and provide a convenient interface which is generally adequate.

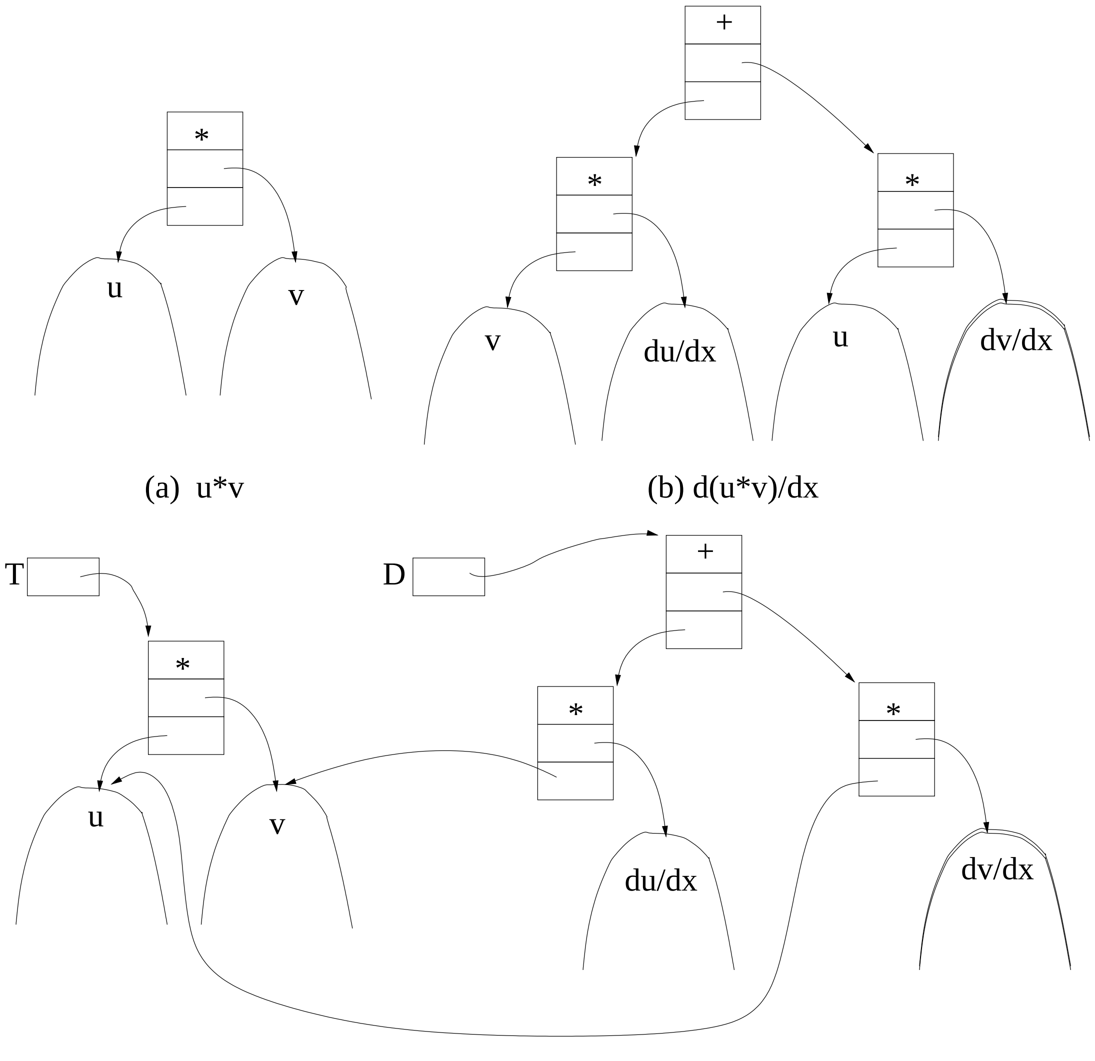
However, if you need to manage the memory yourself, you need to figure out what kind of sharing you want to allow. In Section **??** we gave a solution in which we ensured that each allocated object is pointed to by exactly one pointer. As a result, we can tell fairly easily when the allocated object is no longer needed and can be returned to the heap (Section **??**). This is in fact the memory management idea used in STL, but it executes behind the scenes.

However, the constraint that each object be pointed to by at most one pointer is not always efficient or convenient. Say we have a large tree `T`. Let `L` be a subtree in it. Suppose that we wish to construct another tree `D` which also contains `L`. Then it is natural to share the subtree: we should make the appropriate pointer in `D` point to `L` rather than needing to make another copy of `L` to use as part of `D`. This will require less memory, and potentially save the time required to copy. A similar example actually arises in a program for computing the symbolic derivative of an expression. Consider the rule for differentiating products:

$$\frac{d(uv)}{dx} = v\frac{du}{dx} + u\frac{dv}{dx}$$

When we represent symbolic expressions as trees (Sections 17.1.2 and 20.1), the produce uv will be represented by a tree with u, v being the left and right subtrees, and clearly, u, v will appear as subtrees in the formula for the derivative, specifically the left subtree of the right subtree and the left subtree of the left subtree, see Figure C.1, (a) and (b). So it would be natural to ask: can these two trees, the tree for the original expression and the tree for the derivative, share subtrees, as shown in Figure C.1(c)?

The difficulty in sharing resources is that it is harder to tell when a resource is not needed. If we decide we dont need the tree denoting the original expression any longer, we cannot free the memory used by it, because that memory might be holding parts of the derivative, which we might still need. One way to solve this problem is to use *reference counting*



(a) u*v

(b) d(u*v)/dx

(c) Desired sharing of subtrees

Figure C.1: A function and its derivative

C.1 Reference Counting

The basic idea is to associate a *reference count* with each object, in this case each node of the tree. The reference count of an object is simply the number of pointers pointing to that object, indicating how useful that object is. Ensuring that the reference count is correctly maintained is tricky, but we first describe what we would like to happen abstractly. If we add a new pointer to point to an object, we want one to be added to its reference count. If we remove a pointer, then we want one subtracted. When the count of an object `X` drops to zero, we decide that `X` is no longer useful, and so we return the memory of `X` to the heap. Note that `X` might itself be pointing to an object `Y`. In this case we know that the pointer to `Y` out of `X` will no longer be of use. So the reference count of `Y` should get decremented. If the reference count of `Y` thus drops to zero, we can return `Y` also back to the heap, and so on.

Coming back to our derivative example, suppose initially we only have the product tree. Suppose the tree is pointed to by a variable `T`. Then every node, including the root is pointed to by 1 pointer. Hence at this point we want the reference counts of all nodes to be 1. Note that we do not associate reference counts with variables such as `T` if they are in an activation record rather than in the heap. For simplicity we will assume that `T` is indeed in the activation record.

Suppose next we create the derivative. Say the nodes of the derivative tree are all on the heap, but its root is pointed to by a variable `D` in the activation record. Suppose we did create the derivative tree such that it shares nodes with the original product tree, as in Figure C.1(c). In this picture, the roots of the subtrees u, v have 2 pointers coming in. Hence after creation we would like the roots of these two nodes to have reference counts of 2 each.

Now suppose we write `T=NULL`. This would make the root of the original expression lose the only pointer it had. So we would decrement the reference count of the root. We would find that the root of the original expression has reference count 0. So we can release the memory of the root back to the heap. This would cause the pointers out of the root (of the original expression) to become useless. Thus we would like the reference counts of the objects they point to to be decremented. Thus at this point the reference counts of the trees of u, v would both become 1. If after this we set `D=NULL` then if our reference counting mechanism is working all reference counts will become 0 and everything would be returned to the heap.

It is not too difficult to implement reference counting manually. To each node we add a reference count data member, and we have listed out the conditions under which this must be incremented or decremented. However, the code is cumbersome, and unless it is designed well, its use will also be cumbersome.

C.2 The template class `shared_ptr`

This class provides a standard solution for reference counting. This class and the class `weak_ptr` are known as *smart pointers* and are available in the Boost Smart Ptr library available from `www.boost.org`. These pointers are also a part of C++11, and can be accessed through the GNU C++ compiler `g++` by supplying the option `-std=gnu++0x`. Since our compiler command s++ really calls g++, the option can also be supplied to s++ to get the functionality. In addition, in your programs you need to include the header file <memory>.

A shared_ptr is really a small structure that contains the real pointer, and of course other data needed to manipulate reference counts. The dereferencing and assignment (and other) operators are overloaded for shared_ptrs so that they refer to the real pointer contained inside and also do the bookkeeping needed for reference counting. Thus in many ways a shared_ptr is like an ordinary pointer, and can be used almost as conveniently.

Each shared pointer has a member function use_count which returns the reference count of what the shared pointer points to. Note that in the context of shared pointers, we define the reference count of an object to be the number of shared pointers pointing to it. You need not concern yourself with exactly where the reference count is stored; just rely on the guarantee provided to you.

C.2.1 Synthetic example

Figure C.2(a) shows an example of use of shared pointers, and the output produced when the program is run.

The first group of statements creates and initializes two shared pointers s1, s2 to two instances of A allocated on the heap. When each instance is created, the constructor of A prints the address of the instance. In our execution these happened to be respectively 0x804c008 and 0x804c030. Each object A contains a shared_ptr to another A object. The last statement of the group sets s1->aptr to s2. This has the effect that now s1->aptr points to whatever s2 was pointing. After this we print the reference counts. As you can see s1 points to 0x804c008, and nothing else points to it. However, s2 as well as s1->aptr point to the same instance 0x804c030. Thus the reference count of s1 is 1, and those of the other two are both 2.

In the second group of statements we set s2 to point to a new instance on the heap. The creation causes its address 0x804c058 to be printed. Note that s2 was earlier pointing to 0x804c030, so we should expect its reference count to drop by 1. This indeed happens. Now the three pointers s1, s2, s1->aptr are pointing to unique objects, and the reference counts 1 1 1 are printed for them.

In the third group we set s1=NULL. Since s1 was earlier pointing to 0x804c008, its reference count which was 1 should drop to 0. This should cause this object to be deleted and returned to the heap. This indeed happens, the destructor prints a message saying this. Note further that when 0x804c008 is returned, the contained pointer s1->aptr is no longer valid. Thus the reference count of the object 0x804c030 pointed to by it should also become 0, and that should get destroyed. This also happens, as shown by the statement printed by the destructor. At the end of this group we print the reference counts of s1 and s2 only, since s1->aptr is now invalid. These indeed come out as 0 1, which is correct because s1 is NULL and s2 indeed points to 0x804c008, and is the only one to point to it.

The last print statement indicates that 0x804c008 also gets deleted. This happens because when the program exits the scope, delete commands are issued on all local variables of the current activation frame. Thus a delete command is issued on s1, s2. Provided a shared pointer is non NULL, a delete on a shared pointer causes the reference count of the pointed object to decrement, and if it decrements to 0 to delete that object as well. This is exactly what happens!

C.2.2 General strategy

The preceding discussion might suggest the following strategy for managing memory using `shared_ptr`:

1. First write the program without worrying about memory management, i.e. use ordinary pointers and allocate memory but do not worry about returning it.

2. Replace every pointer which can potentially point to heap allocated memory with a `shared_ptr`.

3. Initialize/assign to `shared_ptr` only by a `new` object, or by the value of another `shared_ptr` or by `NULL`. Note that as in the preceding program, you cannot write `shared_ptr<A> s1 = new A;`, but you must explicitly convert the pointer to a `shared_ptr`.

This strategy works well sometimes. For example, it works quite well for the derivative finding program. We discuss this in Section C.2.3.

This strategy may not work, for several reasons. One possibility is that you may have pointers for which rule 3 above cannot be applied, because originally they were being used to hold `new` addresses as well as addresses of variables in activation frames. In this case you will need to reorganize your program.

However, any implementation of reference counting (including that provided by `shared_ptr`) has one fundamental limitation: if your pointers form cycles, then you will have memory leaks even with `shared_ptr`s. We will see this in Section C.2.4.

C.2.3 Shared pointer in derivative finding

We first develop the code which does not worry about returning dynamic memory. For this we use the representation for trees developed in Section 17.1.2. The code is shown in Figure C.3.

First we have a constructor for constructing terminal or literal nodes. Then we have the constructor for non-leaf nodes. These follow along the lines of Section 17.1.2. Next we have functions `Sum, Prod` and `Lit` that include the `new` operator and thus create the node on the heap. These are convenient to use and the main program at the end uses them. Then we have a `str` function which converts the expression represented by the subtree underneath the node into a string.

Finally, we have the `deriv` function. This uses the definition: the derivative of a sum is the sum of the derivatives, the derivative of a product is as per the rule discussed above, and the derivative of a literal is 1 if and only if the literal is x.

To this code we can apply our recipe for adding in `shared_ptr`s. We only have one kind of pointer in this code: `Exp*`. And as you can see, it is always used to point to heap allocated objects. Also, the pointer structures created in this program will not have cycles. So we do the following:

1. Replace every `Exp*` with `shared_ptr<Exp>`. For this it is better to define `typedef shared_ptr<Exp> spE;`, and then replace `Exp*` by `spE`.

```
#include <simplecpp>
#include <memory>

struct A{
  shared_ptr<A> aptr;
  A(){cout << "Creating A: "<< this << endl;}
  ~A(){cout << "Deleting A: "<< this << endl;}
};

int main(){
  shared_ptr<A> s1(new A), s2(new A);                     // Group 1
  s1->aptr = s2;
  cout  << s1.use_count() << " " << s2.use_count() << " "
        << s1->aptr.use_count() << endl << endl;

  s2 = shared_ptr<A>(new A);                             // Group 2
  cout << s1.use_count() << " " << s2.use_count() << " "
       << s1->aptr.use_count() << endl << endl;

  s1 = NULL;                                             // Group 3
  cout << s1.use_count() << " " << s2.use_count() << endl << endl;
}

---------------------- Output produced --------------------------

Creating A: 0x804c008
Creating A: 0x804c030
1 2 2

Creating A: 0x804c058
1 1 1

Deleting A: 0x804c008
Deleting A: 0x804c030
0 1

Deleting A: 0x804c058
```

Figure C.2: Program to test shared pointers and its output

```
#include <simplecpp>

struct Exp{
  Exp* lhs;
  Exp* rhs;
  char op;
  string value;
  Exp(string v) : value(v) {lhs=rhs=NULL; op=’A’;}
  Exp(char o, Exp* l, Exp* r) : lhs(l), rhs(r), op(o) { value="";}
  static Exp* Sum(Exp* l, Exp* r){ return new Exp(’+’, l, r);}
  static Exp* Prod(Exp* l, Exp* r){ return new Exp(’*’, l, r);}
  static Exp* Lit(string v){return new Exp(v);}
  string str(){
    if (op == ’A’) return value;
    else  return "(" + lhs->str() + op + rhs->str() + ")";
    }
  Exp* deriv(){
    if(op == ’+’) return Sum(lhs->deriv(), rhs->deriv());
    else if(op == ’*’) return Sum(Prod(lhs->deriv(), rhs),
  Prod(rhs->deriv(), lhs));
    else return Lit(value == "x" ? "1" : "0");
  }
};

int main(){
  Exp* e = Exp::Sum(Exp::Lit("x"), Exp::Prod(Exp::Lit("x"), Exp::Lit("x")));
  cout << e->str() << endl;
  Exp* f = e->deriv();
  cout << f->str() << endl;
}
```

Figure C.3: Mini symbolic differentiation program

2. Check assignments to `Exp*` variables. If other `Exp*` variables were being assigned, then therefore there should be no problem because both are converted to `spE`. However, there can be `new` expressions that were assigned to `Exp*` variables. These now have to be explicitly converted to `spE` type. So you will have to do this.

With these two steps, your program should work. Add code to the constructors to observe when they are called, and add destructors so that you know when they are called as well. You should observe that while taking the derivative of a product `uv` the expressions for `u` and `v` are not copied. So they must be shared. You can also see that the nodes are destroyed when the program terminates. So there is no memory leak either.

C.2.4 Weak pointers

Consider a program fragment that uses the `struct A` from Figure C.2:

```
int main(){
  shared_ptr<A> s1(new A), s2(new A);
  s1->aptr = s2;
  s2->aptr = s1;
  s1 = NULL;
  s2 = NULL;
}
```

At this point s1 If you execute this, you will merely get:

```
Creating A: 0x804c008
Creating A: 0x804c030
```

Even when the program finishes, you will not get any deallocations to happen, as you did in the last line of the output of Figure C.2. Let us trace the execution to see why. Clearly, the creation of `s1,s2` caused the messages about creating `A` to be printed. After that, the instruction `s1->aptr = s2; s2->aptr = s1;` causes `s1,s2->aptr` to point to `0x804c008`, and `s2,s1->aptr` to point to 0x804c030. Thus the reference counts of all 4 shared pointers are 2. Now consider what happens when we set `s1 = NULL;` – one reference to 0x804c008 goes away. But it still has 1 reference, and hence no `delete` happens. When we set `s2 = NULL;` next, – one reference to 0x804c030 goes away. But this also has one reference. Thus even after we set `s1, s2` to `NULL`, the objects at `0x804c008` and `0x804c030` continue to have reference count 1, the pointer inside the first contributes the count to the other and vice versa. But our program cannot access these objects, and they havent been returned to a heap: so we have a memory leak.

C.2.5 Solution idea

This problem can only be solved using so called the class `weak_ptr` in conjunction with `shared_ptr`.

The basic idea is to break every pointer cycle by putting one `weak_ptr` in it. A `weak_ptr` is a pointer which does not increment the reference count. However, if the object pointed to by the weak pointer is deleted, then the weak pointer becomes NULL. So whenever you wish to traverse a weak pointer W, you can first check if *W is not NULL and only then traverse. If you are working in a setting in which there are multiple threads, then you might need to *lock* the pointer first.

C.3 Concluding remarks

Managing heap memory in C++ is an evolving field. As a novice programmer, your needs will probably be met by the classes in STL. If for some reason you need to go beyond that, ideas such as shared_ptr (and also weak_ptr if necessary) will likely be adequate. There is work on so called *garbage collection* strategies, but that is beyond the scope of this book.

Appendix D

Libraries

The term *library* is used to refer to a single file which collects together several object modules. Suppose you have constructed object modules `gcd.o, lcm.o`. Then you can put them into a single library file. On unix, this can be done using the program `ar`, and it is customary use the suffix `.a` for library (archive) files, and so you might choose to call your library `gcdlcm.a`. This can be done by executing

```
ar rcs gcdlcm.a gcd.o lcm.o
```

Here the string `rcs` indicates some options that need to be specified, which we will not discuss here. This command will cause `gcdlcm.a` to be created.

When compiling, you can mention the library file on the command line, prefixing it with a `-L`; modules from it will get linked as needed. In fact you could send the file to your friends who wish to use your `gcd` and `lcm` functions, along with an header file, say `gcdlcm.h`, which contains the declarations of `gcd` and `lcm` (but not the definitions). This is the preferred mechanism for sharing code. Note that your friends will not be able to easily know how your functions work, because you need not send them the corresponding `.cpp` files.

D.0.1 Linking built-in functions

You can now guess how built-in functions such as `sqrt` or are linked to your program. They are in libraries, which `s++` supplies when needed! Commands such as `sqrt` are contained in the `cmath` library that is supplied as a part of C++. Our compiler `s++` automatically includes the corresponding library while it compiles your programs. Of course, this is not the entire story – you need to have the prototype for `sqrt` and other functions at the beginning of your program.

These prototypes are present in a file called `math.h`, which you can insert into your program by putting the following line at the beginning of your program:

```
#include <cmath>
```

Our compiler knows where to find this file and places it in your program.

But did you remember to include this file? You might remember that you did put in a similar line in your file:

```
#include <graphicsim.h>
```

Because of this the file `graphicsim.h` is picked up from some place known to `s++` and is placed in your file in place of this line. This file contains the line `#include <cmath.h>` which causes the file `cmath.h` to be included!

Appendix E

IEEE floating point standard

First, if the number to be represented is 0, then it is represented by setting all 32 bits to 0. So assume now that the number to be represented is non-zero. We first write the number as $c \times 2^e$. The numbers c, e are respectively called the *significand* and the *exponent.* Here, the significand must be a binary fraction between 1 and 2. We only retain 23 bits after the decimal point (which we should really call the binary point). This will of course entail some inaccuracy. Further, it is expected that the exponent must be in the range -126 and 127. If this condition is satisfied, then the number is represented as follows.

Bit 31	Bits 23-30	Bits 0-22
0 if positive, 1 if negative	$e + 127$	Bits after binary point in c

Notice that if $-126 \leq e \leq 127$, then $1 \leq e + 127 \leq 254$. Thus bits 23-30 are enough to hold the value $e + 127$. Notice further that we are only storing the bits in the fractional part of c, and not its integer part. This is acceptable because the integer part is always 1 we are ignoring the single bit in c before the binary point If the exponent is larger, then the number cannot be represented[1].

[1] But also see Section 13.6.

Appendix F

Reserved words in C++

The following words cannot be used as identifiers.

Appendix G

Less frequently used operators

In this section we will discuss some of the less frequently used operators.

G.1 Bitwise Logical operators

C++ allows bitwise logical operations to be performed on variables of integer types. For simplicity we will only consider operations on the `unsigned` types.

G.1.1 Or

The operator `|` is the bitwise OR operator, i.e. `p | q` returns a number that is obtained by taking the binary representations of `p,q`, and computing the OR of the corresponding bits. Note that the logical OR of two bits is a 1 if and only if at least one of the bits is 1. Here is an example.

```
unsigned int p=10, q=6, r;
r = p | q;
```

This would cause the bit pattern

$$00000000000000000000000000001010$$

for 10 (decimal) to be OR'ed with the bit pattern

$$00000000000000000000000000000110$$

for 6, to get the result

$$00000000000000000000000000001110$$

which is the bit pattern for 14. Thus at the end `r` would be 14.

G.1.2 And

The operator `&` performs bitwise logical AND. Note that the logical AND of two bits is 1 iff both bits are 1. Thus for `p,q` as defined above, if we write:

```
unsigned int s = p & q;
```

`s` would get the bit pattern 00000000000000000000000000000010, which is the bit pattern for 2. Thus at the end `s` would equal 2.

G.1.3 Exclusive or

The operator `^` is the bitwise exclusive OR operator. Note that the logical exclusive OR of two bits is 1 if and only if exactly one of the bits is 1. Thus if we write

```
unsigned int t = p ^ q;
```

the variable `t` would get the bit pattern 00000000000000000000000000001100, which is the bit pattern for 12. Thus `t` would be 12 after the statement.

G.1.4 Complement

Finally the operator `~` is the (unary) bitwise complement operator. The complement of a bit is 1 if and only if the bit is 0. Thus if we write

```
unsigned int u = ~p;
```

the bit pattern for `u` would have 0s wherever `p` had 1s and vice versa. Thus we would have 1s in all positions except the positions of place value 2 and 8. Thus the value of `u` after the statement would be $(\sum_{i=0}^{31} 2^i) - 2 - 8 = 4294967285$.

G.1.5 Left shift

The operator `<<` is used to shift a bit pattern to the left. Thus `x << y` would cause the bit pattern for `x` to be shifted left by the the value of `y`. By this we mean the following operation. We first throw away the most significant `y` bits, and then append y zeros on the right. The resulting bit pattern is the result of the operation. Note that if the most significant `x` are zero, then `x << y` is simply $x2^y$, where x, y are the values of `x` and `y` respectively.

For `p, q` as we defined, i.e. having values respectively `10` and 6, if we write

```
unsigned int v = p << q;
```

the variable `v` would get the bit pattern 00000000000000000000001010000000. This has the numerical value $10 \times 2^6 = 640$.

G.1.6 Right shift

The operator `>>` is used to shift a bit pattern to the right. Thus `x >> y` would cause the bit pattern for `x` to be shifted right by the the value of `y`. By this we mean the following operation. We first throw away the least significant `y` bits, and then append y zeros on the left. The resulting bit pattern is the result of the operation. Note that `x >> y` is simply $x/2^y$ (integer division), where x, y are the values of `x` and `y` respectively.

For `v` as we defined above, i.e. having value respectively `640`, if we write

```
unsigned int w = v >> 2;
```

the variable `w` would get the bit pattern 0000000000000000000000010100000. This has the numerical value 160.

G.2 Comma operator

Appendix H

The stringstream class

The class iostream is used to define objects such as cin and cout on which we can use the extraction operators >> and << respectively to read or write data. The objects are called streams, because data flows in and out of them.

A stringstream is a stream object, but it is constructed out of a string. To use it, you need to include the header <sstream>. This is especially useful in extracting numbers from strings or converting numbers to strings.

As an example, here is a program that takes two double numbers as command line arguments and prints their product.

```
#include <sstream>
int main(int argc, char *argv[]){
  double x,y;
  stringstream(argv[1]) >> x;
  stringstream(argv[2]) >> y;
  cout << x*y << endl;
}
```

In this we have used the stringstream functionality provided in C++, by including <sstream>. The function stringstream takes a single argument s which is a character string, and converts it to an input stream (such as cin). Now we can use the >> operator to extract elements. Thus stringstream(argv[1]) >> x; would extract a double value from the second word typed on the command line. Similarly a double value would be extracted into y from the third word. Thus if you typed

```
a.out 4 5e3
```

The answer, 20000 would indeed be printed.