Introduction to Programming 1

Only study guide for

COS1511

written by Ken Halland

School of Computing

University of South Africa

COS1511/1/2012-2013

ACKNOWLEDGEMENTS

This study guide for COS1511 was reviewed and approved by the following team:

ROLE	NAME
Director, School of Computing	Lessing Labuschagne
Academic field specialist	Ken Halland
Graphic cover design	Ella Viljoen
Photographs	Ilze Botha, Shutterstock
Layout format	Ken Halland
Critical readers	Drina du Plessis
	Marthie Schoeman
	Petra le Roux
Page layout	School of Computing
Printed and published by the	University of South Africa, Muckleneuk, Pretoria

Contents

Preface

About the guide

The main objective of this study guide is to provide an introduction to programming for people who have never worked with a computer before.

The programming language used in the guide is C++. The main reason for this choice is that C++ is the main language used at Unisa for teaching programming. Since C++ is generally not considered a good language for learning programming, we have selected a subset of the C++ language to cover all the most important concepts for this difficult realm of expertise.

So, even if you master all the work in this guide, you won't really be able to call yourself a C++ programmer. However, our intention is to cover an orthodox (correct) subset of C++ to provide a solid foundation for learning the more complex aspects of C++. In particular, C++ is an object-oriented programming language. Although we do not cover object-orientation in this guide, our intention is to lay a foundation for object-oriented programming.

There are differing opinions on how one should learn object-orientation. Some say that objects and classes should be introduced from the start. We believe however that object-orientation consists of many difficult concepts that need to be understood at the same time. Since this guide is intended to provide a gentle introduction to programming, we have taken a different approach. We cover the building blocks that you should need to be able to solve bigger programming problems. We purposefully do not require you to write any large and complicated programs. For that, you need a methodology (either traditional structured programming or object-oriented programming) to be covered in a separate course.

We hope and trust that, having worked through this guide, you will be excited about programming and motivated to continue with studies in Computer Science.

The approach in the guide is quite novel (and fun), we believe. Each lesson contains one or more main activities, each of which covers one or two basic programming skills. The main activities are generally pitched at a level just above where the reader is expected to be. In other words, the main activities generally require the use of a skill which hasn't been explained properly in the guide up to that point. This works as follows:

Every main activity has a fully worked out solution. Between the main activity and its solution, a number of subactivities are given. These are intended to introduce and give practice in the skills required to complete the main activity. The idea is to continually stretch the abilities of the reader, but to allow the reader to determine the size of the steps.

Each lesson ends with a summary of all the aspects covered in it, as well as a number of exercises for applying the skills that should have been mastered.

How to use the guide

Firstly, you must have access to a computer, and you must sit down and use it as you work through this guide. There is no point whatsoever in reading through or trying to study from this guide without doing the activities practically. Programming is a practical skill (like learning to play tennis) and there is only one way to learn and that is to do it (as there is no way you can learn to play tennis without doing it!)

The computer icon indicates that we strongly recommend that you do the indicated exercise on a computer.

Since the guide is an introduction to programming using C++, your computer will need a C++ compiler and IDE installed on it. This guide does not include instructions for installing and using this software. You

should have received a CD-Rom with this guide containing (amongst other things) a C++ compiler, an IDE, as well as instructions on how to install and use them.

Although this guide has been written particularly for the MinGW port of Gnu C++, other C++ compilers (e.g. MS Visual C++ or Borland C++) could be used without too many problems. The main differences will be the error messages displayed for syntax and run-time errors (mentioned mainly in Part I).

Apart from the guide and a computer with a C++ compiler installed on it, you will need a notebook (referred to in the guide as your study notebook). Many of the subactivities, especially in the first few lessons, require you to write down an answer in your study notebook. We would encourage you to do so to gain the maximum benefit from the guide.

The notebook icon indicates that we strongly recommend that you try and answer the question in your study notebook.

The variable diagrams we use in Lessons 5, 10 and 23 are intended to illustrate how the values of variables change as a program is executed. Marthie Schoeman has developed a series of tutorials to animate these variable diagrams, not only for these lessons, but also to show how the values of variables change during other programming structures, like loops. These tutorials are available separately from this study guide.

The tutor icon indicates that we recommend that you load the appropriate tutorial (with the corresponding activity or subactivity number) and watch the effect of the program statements on the variables.

As mentioned above, the main activities are generally pitched just above the skills which have been covered up to that point in the guide. We recommend that you start each main activity, nonetheless, even when you don't know how to work out the whole solution. This will not be a waste of time, because after working through the subactivities that lead up to the solution of the main activity, you will have to complete the main activity in any case. By having started it, you will also have a greater receptiveness to the skills covered in the subactivities.

In general, be prepared to take risks and make mistakes. (Imagine if you were terrified of ever missing the ball when learning to play tennis. I am sure that it would take you a lot longer to learn to play than if you were just to relax and enjoy yourself.) It is virtually impossible to break a computer by making a mistake. It is possible to lose your work, with the result that you have to do it all over again, however, but then there is no way to learn to save your work than to have made that mistake a few times!

Finally, try to keep track of what you learn. Programming is about designing general methods for solving general classes of problems. Specific problems are given in the guide, to which specific solutions are given, but it is essential to understand the general structures for solving problems. Only understanding a particular solution to a particular problem is not much good. Your aim should be to learn to solve any programming problem, not just specific ones. So make notes for yourself in your study notebook and in the margins of the guide, and try to express the things you have learnt in general terms.

Acknowledgements

This study guide is based on two guides written by John Barrow and Helene Gelderblom for previous versions of the modules COS111U and COS112V at Unisa (up to 1999). We have used the activity/subactivity model, as well as some of the ideas in those guides (which were originally for learning programming using Pascal.) Biffie Viljoen, Drina du Plessis, Marthie Schoeman and Petra le Roux helped with some of the new ideas, and with proof-reading. Annelize de Villiers also helped proof-reading and with some of the exercises at the end of the lessons.

Ken Halland

Pretoria, June 2011

Part I

Starting to program

Unlike a toaster or a fridge which can only do very limited things, a computer can be programmed to do just about anything. You can either get hold of software (a program) that someone else has written, or you can write the program yourself using a programming language.

C++ is a programming language - a language for telling a computer what to do. Like learning a human language, you have to start with the simplest sentences and add to your vocabulary as you go along.

In this part, we see how to tell the computer to do simple input and output, in other words how to display messages on the screen, and how to accept values typed in via the keyboard. We also see how to get the computer to do calculations on numbers, and how to store values in its memory. These represent some of the basic operations or instructions used in most computer programs. Apart from working with integer values, we also look at some other data types, namely floating point numbers, characters and strings. With these instructions and data types, we can already write simple but interesting programs.

Lesson 1

Your first program

Purpose of this lesson

To start this guide, we are going to enter a small program, just to get some idea of what programming is like. It will not be anything spectacular. It will simply display the message `Hello world` on the screen. The program to do this looks as follows:

```
#include <iostream>
using namespace std;
int main( )
{ cout << "Hello world"; return 0; }
```

But before we go on with this program, have you used a computer before, and do you know how to load your IDE and C++ compiler? If you do, please continue with this lesson. However, if you are new to computers or to programming, you will first have to find out how to load your IDE and C++ compiler and how to use it. Then come back to this lesson, and continue with the activities we give below.

Activity 1.a

Type the program shown above in the editor of your C++ IDE. Then compile and run it.

Test yourself

Well done if your program displayed `Hello world` on the screen. You may jump forward to Activity 1.b and continue there.

If your program did not seem to work, or it worked but you would like to know why, continue with the subactivities that follow. They will guide you step by step through entering and running this program.

Subactivity 1.a.i

Typing in your program

Type in the above program. You should be in the editor of your C++ integrated development environment (IDE). Type each line from the program above exactly as it stands, followed by the <Enter> key at the end of the line.

Note that capitalisation is very important. In other words, it is essential that you use lower case for all the letters except the `H` of `Hello`.

Before you proceed, it would be a good idea to save your program now. At the moment, the program you have typed is only in temporary memory called RAM. If the power were to fail at this stage, all your work

would be lost. This text does not give instructions for saving your programs - they should have been provided with your IDE and compiler.

Save the program in a file called **first.cpp**.

Now you are ready to compile and run your program. This will depend on the specific IDE and compiler that you are using, and so is also not covered in this text. You should have received instructions on how to do this with your IDE and compiler.

When you run the program, one of two things will happen. If you entered the program exactly correctly, an output window will appear with `Hello world` displayed in it. Congratulations! You have just got your first program to work.

If there was some mistake, an error message will probably have been displayed in the IDE.

Correcting mistakes

In programming it is very common for a program not to compile correctly, in which case the compiler displays an error message. If this happens to you, there has been a typing mistake.

Everybody makes mistakes now and then, even the world's best programmers. With practice, programmers become very good at finding and fixing their mistakes! Compare the program you have entered on the screen in front of you very carefully with the program as it is printed out at the beginning of this lesson. It must be exactly the same, because computers are very fussy.

When you find a mistake, use the arrow keys on the right of the keyboard to place the flashing cursor on the mistake. If anything is missing, type in the missing part.

Sometimes, when you press an arrow key, the flashing cursor does not move, and you get a number on the screen instead. If this happens, press the <Num Lock> key towards the right hand side of the top row of the keyboard to change the number pad from numbers to the cursor control functions.

Once you have corrected any mistakes, compile and run your program again. If the program is now correct, the output will appear as we described above. However, if you get an error message again, there is still a mistake that you must correct.

Keep checking the program and correcting the mistakes until you get it running properly, and then continue with the following subactivity.

Subactivity 1.a.ii

As you have probably guessed, we can change the program to display something other than `Hello world`.

Can you work out how to change what the computer will display on the screen?

Before reading any further, try to make the computer display:

```
How are you?
```

Subactivity solution

To change what the computer displays, use the arrow keys to place the cursor on the `H` in `"Hello world"` in the program. Press the <Delete> key to delete the characters `Hello world` and type `How are you?` in their place. Note that you will have to hold down the <Shift> key to get the question mark. Your new program should look like this:

```
#include <iostream>
using namespace std;
int main( )
{ cout << "How are you?"; return 0; }
```

Compile and run your edited program. The output `How are you?` should appear unless there is an error. If there is an error, correct it as described above.

Now experiment with some other phrases of your own.

Subactivity 1.a.iii

Making mistakes on purpose

In the next few subactivities, we are going to make some deliberate mistakes. Why would anyone want to do that? It is because we want to learn more about the computer. We want to understand what the computer does when we make mistakes, and we will come to see that although the computer is very fussy, it is also very patient. Experimenting like this will help us in future to solve some problems we may have more easily.

Let us see what happens when we make a typing mistake and leave out the second curly bracket.

Use the arrow keys to place the cursor on } in the program. Press the <Delete> key to delete it. Your program should now look like this:

```
#include <iostream>
using namespace std;
int main( )
{ cout << "How are you?"; return 0;
```

Compile the program.

What happens? An error message appears - something like:

parse error at end of input

Fix the problem by putting the } back in. Now the program should compile correctly again and you are ready to try the next subactivity.

Subactivity 1.a.iv

Let us see what happens when we make the `c` of `cout` an upper case letter.

Use the arrow keys to place the cursor on the `c` in the `cout` statement in the program. Delete the lowercase `c`, and type an uppercase `C` in its place. Your program should now look like this:

```
#include <iostream>
using namespace std;
int main( )
{ Cout << "How are you?"; return 0; }
```

Compile the program.

What happens? An error message appears - something like:

Cout undeclared

Remember that C++ is case sensitive. In other words, the (upper or lower) case of all the instructions in C++ are very important. If you type one or more characters in the wrong case, C++ will not recognise the instruction.

Of course, C++ is not case sensitive about strings of characters that form messages. For example, we could change the program to display the message `HOW ARE YOU?` and C++ would do that quite happily. But that's just because the message is not one of the instructions that C++ has to recognise. All the other parts of the program are instructions, and they must be in the correct case.

Subactivity 1.a.v

Try making some other errors. For example, you could change the first double quote mark to a single quote mark to get:

```
{ cout << 'How are you?"; return 0; }
```

If you try to compile this now, you will get an error message like:

unterminated string or character constant

We know what the mistake is, but sometimes it is difficult to work out because the compiler's error messages are often not very helpful. Replace the single quote with a double quote, and the program should compile correctly once more. (Any language, whether it is a computer language or a human language, has rules which one must follow if others are to understand what you mean. These rules are called *syntax rules*. We will look at the syntax rules of C++ a bit more carefully later.)

Subactivity 1.a.vi

Just for practice, let us make one more mistake. A very common mistake made by C++ programmers is to use the >> operator when they should use <<. Use the arrow keys to move to the << operator. Delete these two characters and replace them with >> by pressing > twice. What happens now if you try to compile?

You get an error message like:

no match for `ostream & >>`

This is a particularly common mistake because there are situations (which we will see in Lesson 3) where you will need to use >> instead of <<. As a general rule, you can remember that we always use << together with cout. Put the << back in and, for the last time, compile the program.

Activity 1.b

If you have an analysing mind (and you need to develop your analytical skills to be a good programmer!) you should have been somewhat uncomfortable with having been told to type in all these strange instructions without any explanation. You should have been asking yourself what the various parts of the program mean and what they do.

In your study notebook write down a number of questions about the code. Here is an example: it Must the code all be on three lines? In other words, can't I space things out a bit to make things more readable?

Test yourself

You should have been able to write out a number of questions, like what `#include` means, and what the curly brackets are for, etc. Even though you managed to write down many questions, we recommend that you work through the subactivities below which attempt to explain and answer (at least some of) the questions that you asked, and maybe some that you should have asked but didn't!

Subactivity 1.b.i

Firstly we want to answer the example question that we posed above, namely whether the program needs to be typed on three lines.

If you don't have the program that you typed in Activity 1.a still on the computer screen, you will need to load (or open) it in the editor of the IDE again. (Once again, this text does not give instructions on how to do this.)

Now edit the program to look like this:

```
#include <iostream>
using namespace std;

int main( )
{
   cout << "Hello world";
   return 0;
}
```

To get the additional lines in the code, you need to move the cursor to the place where you want another line and press the <Enter> key. In other words, go to the end of the first line and press <Enter>. Then go to the space just after the open curly bracket and press <Enter>, etc.

Note the following differences between this program and our previous version:

There is an open line between `using namespace std;` and `int main( )`. Furthermore, the statements `cout << "Hello world";` and `return 0;` are each on a line on their own. The opening and closing curly brackets are each placed on a line of their own.

We also indented the two statements `cout << "Hello world";` and `return 0;`. We did this by pressing the <Tab> key at the beginning of these lines.

You should be able to compile this program again without any errors. Try it now. If the compiler does give error messages, check the program carefully to see whether any other mistakes have crept in.

So, the answer to the question is NO, the program need not be on three lines. In fact the programming style in our first program was not very good. We will talk about programming style throughout this text because it is important that you develop a habit of writing your programs in a neat and readable way.

Leaving open lines, putting separate statements on separate lines, and indenting code between curly brackets is part of good programming style. As you can see, the compiler doesn't mind whether a program is written in bad programming style, but for us humans, it is more difficult to read and understand what the various parts of a program do when it is written in bad style.

Subactivity 1.b.ii

One question that you probably asked is what the statement `#include <iostream>` at the beginning of the program means. This is a special instruction that we always put at the top of the code to tell the compiler to include the `iostream` header file.

C++ contains a number of header files that make programming easier. We will need to include other header files in programs that we write later on (in Lesson 7 and further on). Header files basically define a number of additional instructions that we can use in our programs to make them simpler and neater. A particularly useful one is `iostream`. The `io` in `iostream` stands for Input/Output and this header file allows us to use the `cout` instruction to output messages to the screen. To see this, delete the entire line `#include <iostream>` and compile your program again.

It is interesting to note that the compiler doesn't complain that the statement `#include <iostream>` is missing, but rather that it doesn't recognise `cout`. This is because `cout` is defined in the `iostream` header file.

(Another question that you might have asked is what the angle brackets on either side of `iostream` are for. These are to indicate that `iostream` is a header file provided with standard C++, and are essential. If you leave them out, the compiler will be unhappy.)

Subactivity 1.b.iii

What does `using namespace std;` mean?

Some C++ compilers allow you to leave this out. If you do leave it out, you should change the program as follows:

```
#include <iostream>

int main( )
{
  std::cout << "Hello world";
  return 0;
}
```

As explained above, `cout` is defined in the `iostream` header file. All the names defined in it belong to a *namespace* called `std`. If you don't use `using namespace std;` in your program, then you should prefix every name that belongs to the `std` namespace with `std::`.

Subactivity 1.b.iv

What does `int main( )` mean?

This is the header for the main function of the program. In fact, the main function includes the two curly brackets and all the statements in between. This line on its own is called the header of the main function.

Every C++ program must have a main function.

The reserved word `int` in the function header specifies the return type of the main function. You need not worry about this now. It will become clearer when we discuss return types in the lessons about defining functions. Without going into a complicated explanation, it is related to the statement `return 0;` The number `0` is called an integer, and `int` tells the compiler that the main function will be returning an integer value.

Subactivity 1.b.v

What are the semicolons for?

In C++, a semicolon is called a *statement terminator*. There are two statements in the body of the main function. (The body of the main function - as opposed to its header - includes the two curly brackets and all the statements in between.) Each statement must be terminated by a semicolon. Try removing either of the semicolons and compiling your program again. The compiler displays an error message like

parse error before return

or

parse error before }

Once again, the compiler only picks up the error when it gets to the statement on the next line. If an error message is unclear, always check the previous line for some mistake like this.

Subactivity 1.b.vi

What does `cout << ...` mean?

The instruction `cout` actually stands for **c**onsole **out**put. It tells the computer to output (i.e. display) what follows the `<<` operator on the console (i.e. on the screen). Note: we pronounce `cout` as "see-out".

Activity 1.c

Add a comment statement to the beginning of the program given in Subactivity 1.b.i to explain what the program does.

> **Test yourself**
>
> This is a somewhat unfair task, because we haven't explained what a comment statement is, nor how to specify one.
>
> A comment statement is text that we add to a program that has no effect on the running of the program but that is generally used to explain what the program (or part of the program) does.
>
> The following subactivities illustrate how to add a comment statement to a program.

Subactivity 1.c.i

Edit the program of Subactivity 1.b.i to look like this:

```
#include <iostream>
using namespace std;

int main( )
{
```

```
  //Display a message
  cout << "Hello world";
  return 0;
}
```

Remember: To insert a line, go to the end of the line where you want a new line and press <Enter>. Then type `//Display a message`.

Compile and run the program to see what effect it has.

Discussion

As you should have noticed, the comment statement has no effect whatsoever. The compiler simply ignores the entire line.

You should be able to do the main activity now, but before you do, try the following subactivity for an interesting surprise.

Subactivity 1.c.ii

Type `//` at the beginning of the line `cout << "Hello world!";`. Before recompiling and running the program, try to predict what effect this change will have on the program.

Subactivity solution

There is a definite change to the output of the program - nothing is displayed! This is because the compiler now ignores the `cout` statement that displays the message on the screen. It treats the entire statement as a comment.

Remove the changes you made for Subactivities 1.c.i and 1.c.ii (by deleting all the characters you added) and do the main activity now. Put the comment right at the beginning of the program.

Activity solution

```
//Displays Hello world on the screen
#include <iostream>
using namespace std;

int main( )
{
  cout << "Hello world";
  return 0;
}
```

Your comment need not have been identical to ours. Any descriptive sentence that you could read at a later date and immediately see what the program does would have been good enough.

Discussion

In this activity we added what is called a comment statement, i.e. a statement starting with the two characters `//`. The compiler completely ignores the rest of the code on a line that has these two characters.

We always put a comment statement at the beginning of our programs to explain what they do. This is another aspect of good programming style. We will see other uses of comment statements later in this study guide.

Activity 1.d

Write a program to display the following on the screen:

```
Hello, everybody!
My name is YourName.
Goodbye.
```

Where *YourName* appears, insert your own name.

Test yourself

Think about what is different about the output that the program must display. Then think what you need to do differently to write this program.

You could use one `cout` statement and use many space characters (as many as needed) to make sure the second line appears on the next line of the screen, etc.

There is a better way however, and that is to use a separate `cout` statement for each line together with `endl` (hint, hint!).

Subactivity 1.d.i

Predict what the output of the following program will be:

```
//Displays Hello world on the screen
#include <iostream>
using namespace std;

int main( )
{
  cout << "Hello";
  cout << "world";
  return 0;
}
```

Subactivity solution

You might have been surprised at the output, namely

```
Helloworld
```

Note that the two words are displayed on the same line with no space in between. So this subactivity simply shows that you can use more than one `cout` statement in a program. The problem still remains how to get different parts of the output on separate lines.

Subactivity 1.d.ii

Insert `<< endl` at the end of the first `cout` statement, i.e. just before the semicolon at the end of the line. Predict what the output of the program will be now.

Subactivity solution

Now the output is

```
Hello
world
```

Discussion

Apart from getting the `cout` statement to display a message on the screen, we can get the computer to display output on separate lines on the screen. For this we use the *stream manipulator* `endl`.

The word `endl` stands for **end l**ine, and is also defined in the `iostream` header file. When used with `cout`, it makes the output continue on the next line. It is particularly useful if you want to display more than one message (each with a separate `cout` statement) and you want each message to appear on a new line.

Activity solution

```
//Displays chatty messages on the screen
#include <iostream>
using namespace std;

int main( )
{
   cout << "Hello, everybody!" << endl;
   cout << "My name is YourName." << endl;
   cout << "Goodbye." << endl;
   return 0;
}
```

Important points in this lesson

Programming concepts

In this lesson, we typed in and compiled some simple C++ programs that displayed short messages on the computer screen.

We saw that the computer is very fussy about how a program is entered. If there are any syntax errors, or the wrong case is used, the compiler gives an error message and we must then correct these mistakes before it will compile correctly.

The final "Hello world" program that we wrote was:

```
//Displays Hello world on the screen
#include <iostream>
using namespace std;

int main( )
{
  cout << "Hello world" << endl;
  return 0;
}
```

There are many aspects of this program that will become clearer as we go on. The most important to understand at this point are `cout`, `<<` and `endl`.

The word `cout` stands for **c**onsole **out**put and is defined in the `iostream` header file. It tells the computer to output (i.e. display) what follows the `<<` operator on the console (i.e. on the screen).

The word `endl` stands for **end l**ine, and is also defined in the `iostream` header file. When used with `cout` and `<<`, it makes output continue on the next line. It is particularly useful if you want to display more than one message (each with a separate `cout` statement) and you want each message to appear on a new line.

We can describe the structure of a simple C++ program as:

```
//DescriptiveComment
#include <StandardHeaderFile>
using namespace std;

int main( )
{
   StatementSequence;
}
```

Programming principles

As far as programming style goes, we saw how to use comments, open lines between statements and indentation to make our programs more readable.

Exercises

Exercise 1.1

Compare the description of the program structure that we have just given with the program we gave when describing the purpose of this lesson. What part of the program corresponds to *DescriptiveComment*? What parts correspond to *StandardHeaderFile* and *StatementSequence*?

Exercise 1.2

Write a program to display the following poem[1] on the screen.

```
Twinkle, twinkle, little bat!
How I wonder what you're at?
Up above the world you fly,
Like a tea-tray in the sky.
```

[1] from *Alice's Adventures in Wonderland* by Lewis Carroll

Lesson 2

Integers

Purpose of this lesson

In Lesson 1, we saw how to write a program to display short text phrases on the screen. In this lesson, we are going to do calculations on numbers and display their results on the screen.

Activity 2.a

Write a program to calculate and display the product of the first five positive integers. The program should also display an explanatory message with the answer, namely

```
The product of 1 to 5 is 120
```

Test yourself

If your program contained the single `cout` statement:

```
cout << "The product of 1 to 5 is 120" << endl;
```

then you cheated! You were meant to *calculate* the product, not just display it. If you managed to write the program to calculate the product without looking at the subactivities (and solution to the activity) below, you probably have programmed a computer before. If not, do Subactivity 2.a.i below and then attempt the activity again. If you still are not sure how to complete the activity, Subactivity 2.a.ii should cover all you need to know to do so.

Subactivity 2.a.i

Before typing in, compiling and running the following program, predict what it will display:

```
//What does it do?
#include <iostream>
using namespace std;

int main( )
{
  cout << 1 + 2 + 3 + 4 << endl;
  return 0;
}
```

Subactivity solution

We are going to be a bit nasty and not tell you what is displayed to encourage you to type the program in and get it to work to check whether your answer is correct.

All that we are prepared to say is that the program calculates and displays the sum of the first four positive integers.

Subactivity 2.a.ii

In your study notebook, write down what you think the output of the following program will be:

```
//Displays the product of 371 and 194
#include <iostream>
using namespace std;

int main( )
{
  cout << "The product of 371 * 194 is ";
  cout << 371 * 194 << endl;
  return 0;
}
```

You might need a pocket calculator to make your prediction.

Subactivity solution

The output will be

*The product of 371 * 194 is 71974*

You might like to type in the program yourself and check whether this is correct.

Discussion

Note the following interesting aspects of the program and its output:

- The first occurrence of the expression `371 * 194` (that appears in the message, between the quotation marks) is not evaluated and is displayed as is, whereas the second occurrence (that does not have quotation marks around it) is evaluated.
- Even though there are two `cout` statements, the output is all displayed on a single line. This is because there is no `<< endl;` at the end of the first `cout` statement. You might like to test this by adding `<< endl;` to the end of this line and compiling and running the program again.
- The output contains a space between the word `is` and the result, namely `is 71974`. This is because there is a space character between `is` and the closing quote character in the program.

You should be able to complete Activity 2.a now.

Activity solution

One program to calculate and display the product of the first five positive integers looks like this:

```
//Displays the product of the first five positive integers
#include <iostream>
using namespace std;

int main( )
{
  cout << "The product of 1 to 5 is ";
  cout << 1 * 2 * 3 * 4 * 5 << endl;
  return 0;
}
```

An alternative solution is

```
//Displays the product of the first five positive integers
#include <iostream>
using namespace std;

int main( )
{
  cout << "The product of 1 to 5 is " << 1 * 2 * 3 * 4 * 5 << endl;
  return 0;
}
```

This second solution shows that we can display a string message (using quotes) and the result of a calculation (i.e an expression) using a single `cout` statement. They must just be separated by the `<<` operator.

Activity 2.b

In your study notebook, write down expressions to perform the following calculations:

(i) The product of 12 and 23 plus the product of 34 and 45

(ii) 4 times the sum of 5 and 6

(iii) The difference between 543 and 234

(iv) The difference between 234 and 543

(v) The quotient of 30 and 3 (i.e. 30 divided by 3)

(vi) The quotient of 20 and 3

(vii) The quotient of 10 and 3

(viii) The quotient of the sum of 357 and 468 and the product of -19 and 28

Next to each of these expressions, write the result that you would expect the computer to produce if it was included in a program.

Test yourself

You should be able to do this activity just from your school arithmetic. In particular, multiplication and division are done before addition and subtraction, and you can use brackets to force things otherwise. Note that we use the forward-slash / for division.

Even if you are fairly sure that your answers are correct, we suggest that you do Subactivity 2.b.i before you check your answers in the activity solution below it. You might just have to revise some of your answers!

Subactivity 2.b.i

What will the output of the following program be? Explain how you got to the answer.

```
//Order of operations
#include <iostream>
using namespace std;

int main( )
{
  cout << 1 - (2 + 3 * 4) / 5 << endl;
  return 0;
}
```

Subactivity solution

The output will be -1. If you don't believe it, edit the program you typed in earlier to look like this, and test it.

This answer is calculated in the following way:

The brackets are done first, in other words 2 + 3 * 4:

3 * 4 is calculated before 2 is added because multiplication is done before addition.

3 * 4 is 12

2 + 12 is 14 so the result of the expression in round brackets is 14.

14 / 5 is done before anything is subtracted from 1 because division is done before subtraction. In other words, we don't do 1 - 14 and then divide the answer by 5!

Surprise, surprise 14 / 5 is 2 because we are working with integers, and when one integer is divided by another, the remainder (i.e. the fraction part) is thrown away.

1 - 2 is -1.

We often use another format for showing the order of calculations:

```
1 - (2 + 3 * 4) / 5
          |
```

```
1 - (2 +  12  ) / 5
       |
1 -   14        / 5
                |
1 -             2
  |
 -1
```

Note how we repeatedly draw a vertical line under the operator that we evaluate next and replace that operation with its result in the entire expression. We keep doing this until the entire expression is reduced to a single value.

Activity solution

The respective expressions are as follows:

		Comments
(i)	*12 * 23 + 34 * 45*	Round brackets aren't necessary because * is done before +
(ii)	*4 * (5 + 6)*	Round brackets are necessary because * is done before +
(iii)	*543 - 234*	
(iv)	*234 - 543*	Integers include negative numbers
(v)	*30 / 3*	
(vi)	*20 / 3*	The remainder is thrown away
(vii)	*10 / 3*	The remainder is thrown away
(viii)	*(357 + 468)/(-19 * 28)*	The first pair of round brackets is necessary to force the addition before the division. The second pair of round brackets is necessary to force the multiplication before the division since they are normally done from left to right.

We haven't given the answers on purpose to encourage you to check the results you expected on the computer. Make sure that you understand numbers (v), (vi), (vii) and (viii) in particular.

Important points in this lesson

Programming concepts

We have learnt the following things in this lesson:

- We can write programs that make calculations on integers. Integers are like the whole numbers since they include 0 and the negative numbers. No fractions are allowed with integers. (In Lesson 6 we will be looking at floating-point numbers, where fractions are allowed.)
- An integer expression is a number of integer values and integer operators (+, -, * and /). Addition and subtraction are done after multiplication and division. If addition and subtraction (or multiplication and division) are done next to one another, they are performed from left to right. Round brackets can be used to force certain subexpressions to be calculated before others to get around these rules.
- The division operator / throws away any remainder (i.e. the fraction part).

- The value of an expression can be calculated and displayed in a `cout` statement. A string can be displayed with the value of such an expression in the same `cout` statement by separating them with the `<<` operator. Any integers and/or operators included in a string (i.e. between quote characters) are not evaluated but are displayed as is. Only expressions without quote characters are evaluated.
- We can also display a message and the value of an expression on the same line using two separate `cout` statements. To do this, we omit `<< endl` at the end of the first `cout` statement. The output of the second `cout` statement will then be displayed on the same line as the output of the first `cout` statement.

We hope you were somewhat irritated by the repetition in exercises (v), (vi) and (vii) in the last activity, particularly if you tested it out on the computer. You would have had to change the program and recompile it each time, just to test how division is performed for different values. In the next lesson we will see how to use variables to make a program more general, i.e. to work with any numbers, not just fixed ones that are hard-coded in the program.

Programming principles

Sometimes it is a good idea to use round brackets in an expression (even when they aren't strictly necessary) to make the order of evaluation clearer.

Exercises

Exercise 2.1

Add round brackets to the following C++ expressions to show the order in which the operators will be evaluated:

(i) 80 / 5 + 70 / 6

(ii) -5 + -4 - -3

(iii) 6 * 7 / 8 * 9

(iv) 1 - 2 + 3 / 4 * 5

(v) -1 + 23 / -4 + 56

Also give the value of each of the expressions.

Exercise 2.2

Write a program that produces the following output:

```
There are 60 seconds in a minute.
There are XXX seconds in an hour.
There are YYY seconds in a day.
There are ZZZ seconds in a year.
```

In place of *XXX*, *YYY* and *ZZZ*, the program should calculate and display the appropriate number of seconds.

Exercise 2.3

Write a program to calculate the remainder of 234 divided by 13. Remember that the / operator throws away the remainder. Hint: divide 234 by 13 and then multiply it by 13 again. The difference between 234 and the result of this will be the remainder.

Lesson 3

Variables

Purpose of this lesson

Variables provide a way to make programs more general. In this lesson we investigate the use of integer variables to allow us to get a program to input numbers from the keyboard, and work with them instead of just using numbers that were coded into the program when it was written.

Activity 3.a

Write a program to input two numbers from the keyboard and determine and display their quotient. The following should appear on the screen while the program is running:

```
Enter two numbers: 23 5
The quotient of 23 and 5 is 4
```

(Note that the numbers 23 and 5 and the space between them are underlined to indicate that these values are entered by the user while the program is running.)

Test yourself

We don't expect you to be able to write this program straight away because you probably don't have the knowledge of how to do so yet. The subactivities that follow explain the various concepts that you need to understand to be able to write this program.

If you have done programming in some other language before, you probably have a fairly good idea of what you need to do, but just need to know how to do it in C++. The first subactivity below should be enough for you to complete the activity problem.

Note that your program should work correctly for *any two numbers* entered by the user.

Subactivity 3.a.i

Consider the following program:

```
//What happens?
#include <iostream>
using namespace std;
```

```
int main( )
{
  int age;
  cout << "Enter your age: ";
  cin >> age;
  cout << "That's a great age to be!" << endl;
  return 0;
}
```

See if you can predict what will happen when this program runs. Then compile and run the program to check whether your prediction was correct.

Also try to explain the purpose of the statements `int age;` and `cin >> age;`. (If you're not sure, guess!)

Subactivity solution

Once again we don't give the output to encourage you to type in and test out the program yourself. Note that the program should pause in the middle with the cursor flashing to indicate that it is waiting for you (the user) to enter a number. Only after you have typed the number and pressed <Enter>, will the rest of the program be executed.

Discussion

The statement `int age;` is called a *declaration statement*. It declares a variable called `age`. A declaration statement sets aside a memory position for a value to be stored and associates a name with that memory position. We can illustrate the situation with a diagram:

The box represents the memory space. We write the name of the variable above the box.

A variable has a name so that we can refer to the value stored in the memory position later in the program. We decided on the name `age` because we intend to store a number representing an age in it.

Note that a variable name has no meaning to the computer whatsoever. You could change the name `age` to `pumpkin` (or whatever you like) and the program would still work perfectly. We use meaningful names so that we and others can read and understand our programs.

By the way, the numbers that we used in Lesson 2 are called *integer literals* as opposed to *integer variables* that we have now learnt about.

The statement `cin >> age;` is called an *input statement*. It is used to obtain a value from the keyboard and store it in a variable, in this case `age`. We indicate this in our variable diagram as follows:

Like `cout`, `cin` is defined in the standard header file `iostream`. It stands for **c**onsole **in**put, and is pronounced as "see-in". The console therefore does not just refer to the screen (as we said in Lesson 1) but also to the keyboard.

Note that we use the `>>` operator with `cin` as opposed to `<<` with `cout`. Programmers often make the mistake of using the wrong operator with either of them. One way to remember is that the arrows indicate the direction of movement of data. In other words, with `cout` the data moves from the program to the console (i.e. to the screen) whereas with `cin` the data moves from the console (i.e. from the keyboard) to the variable in the program.

Note finally that there is a space between the colon and the close quote character in the initial `cout` statement. This ensures that a space is displayed on the screen after the colon for the user to enter the number. You might like to see what the effect is if this space character is removed from the program.

Subactivity 3.a.ii

Not all names that you can choose for variables are legal. C++ has rules for what names are legal for variables and what characters may be used in them. For this subactivity you must investigate what variable names are legal.

After your investigation, you should be able to answer the following questions:

(i) Can numeric characters and any letters of the alphabet be used in a variable name, and in what order?

(ii) What other characters can and can't be used?

(iii) Are there any words that are not allowed to be used?

Use a very simple program like this to do your testing:

```
//Tests variable names
int main( )
{
   int VariableName;
   return 0;
}
```

(Note that we have left out `#include <iostream>` and `using namespace std;` since the program does not use `cin` or `cout`.)

Simply change *`VariableName`* each time and recompile.

Subactivity solution

The answers to the questions are as follows:

(i) Any combination of numeric characters and letters of the alphabet can be used, except that a name cannot start with a numeric character.

(ii) The only other character that may be used in a name is the underscore character _. Names may also start with _.

(iii) Certain words like `int` and `return` can't be used. However, a variable can be declared with the name `cout`, `cin` or `endl`.

If you didn't investigate the names as we suggested above, why not try it now? Type in the little program and test these three rules to convince yourself that they are correct.

Discussion

With respect to point (i), note that a variable name can consist of lower or upper case letters of the alphabet. In fact, you can make a variable all upper case letters if you want. We generally use lower case letters.

About point (iii): C++ has a number of so-called *reserved words*, and you may not use a reserved word for a variable name. Appendix A at the end of this guide contains a list of C++ reserved words. It is interesting to see that `cout`, `cin` and `endl` are not reserved words. They are defined in the `iostream` header file. One might expect the compiler to complain if you declare a variable with one of these names (eg. `int cout;`) and then add `#include <iostream>` and `using namespace std;` to your program. It doesn't. It only complains when you attempt to use the variable, eg. `cout << endl;` The moral of the story is: It's a bad idea to use `cout`, `cin` or `endl` as variable names because it just causes confusion.

The opposite is also true. If you mistakenly use one of these names as if it were a variable (eg. `cout << cout;`) the compiler will not complain. You will just get unexpected results. The moral of this story is, if you get strange results, check whether you haven't made this kind of error.

Subactivity 3.a.iii

Change the program in Subactivity 3.a.i slightly to produce the following output:

```
Enter your age: 29
29 is a great age to be
```

Compile and run your program to see whether it works. Make sure that the number displayed at the beginning of the last line of output is the same as the number the user entered on the first line. Also add a comment line at the beginning of the program explaining what it does.

Subactivity solution

```
//Inputs age and displays it with a message
#include <iostream>
using namespace std;

int main( )
{
  int age;
  cout << "Enter your age: ";
  cin >> age;
  cout << age << " is a great age to be!" << endl;
  return 0;
}
```

Discussion

Note the following about this program:

- We can display the value of a variable in a `cout` statement. Furthermore, it can be displayed together with other strings (and values of expressions) in a single `cout` statement.
- Once again we use a space character between the opening quote and the word `is` in the second output statement to ensure that a space is displayed between the value of the variable and `is` when the program runs.

Subactivity 3.a.iv

Edit the program by removing the declaration statement. In other words, delete the line `int age;`. Try to compile the program again. The compiler will complain about the statement `cin >> age;` with the message

age undeclared

Put the declaration statement back, but in a different place this time, namely between the first output (`cout`) statement and the input (`cin`) statement. Compile the program again to check whether it is OK. It should compile (and run) correctly. Now move the declaration just after the input statement. If you compile the program again, the compiler should give the same error message as before.

To see a similar problem, move the declaration statement back to its original position (at the beginning of the program) and change the output statement to `cout << years << " is a great age to be!" << endl;`. Then try to compile the program again. This time the compiler complains that `years` is undeclared.

Discussion

The point is that we must always declare a variable before we use it or refer to it.

This may appear to be a very simple rule to follow, but often one can make quite subtle errors that are connected to this problem. A common error is to misspell a variable name slightly. For example, say we declare a variable as `aga` (instead of `age`). Then when we refer to `age` later in the program the compiler complains that it is undeclared. We then need to go and fix the name in the declaration. (Alternatively we could change all occurrences of the variable to `aga`, but although that would make the compiler happy and the program would probably work correctly, it would be a less sensible thing to do.)

Another subtle error is changing the case of one or more letters in the name. For example, say we declared a variable as `yourAge` and then referred to it later as `youRage`. Remember that C++ is case sensitive - also for variable names.

Subactivity 3.a.v

See if you can change the program to display the following output:

```
Enter your age: 29
You will be 30 on your next birthday!
```

Once again, make sure that the second number that is displayed is dependent on the number that is entered.

Subactivity solution

```
//Inputs age and displays age + 1 with a message
#include <iostream>
using namespace std;

int main( )
{
  int age;
  cout << "Enter your age: ";
  cin >> age;
  cout << "You will be " << age + 1 << " on your next birthday!" << endl;
  return 0;
}
```

In this program we see that variables and numbers can be used together in an expression. In fact, we can make things as complex as we like. We can use an integer variable anywhere in an expression where we would have used an integer literal.

Subactivity 3.a.vi

Now change the program to display the following output:

```
Enter your mom's age and your age: 50 29
Your mom was 21 years old when you were born.
```

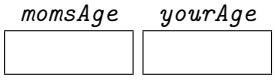
Subactivity solution

```
//Calculates mother's age when you were born
#include <iostream>
using namespace std;

int main( )
{
  int momsAge, yourAge;
  cout << "Enter your mom's age and your age: ";
  cin >> momsAge >> yourAge;
  cout << "Your mom was " << momsAge - yourAge << " when you were born." << endl;
  return 0;
}
```

Discussion

You probably used two (separate) declaration statements on separate lines, eg. `intámomsAge;` and `int yourAge;`. This is not incorrect. We just used a shorthand notation to declare two variables in the same declaration statement, separating the two names by a comma. In fact, you can declare as many integer variables as you like in a single declaration statement.

After the declaration statement(s), we can indicate the state of affairs with the following variable diagrams:



Note once again that we have used descriptive names for the variables. The names describe the values that are stored in the variables. When we want a variable name to consist of more than one word, we generally connect all the words together, and use uppercase for the first letters of all subsequent words. (This is called *camel notation* - because the humps are in the middle!) For example, if we had a variable that we wanted to store the tax before profits, we could call it `taxBeforeProfits`. Note that camel notation is only a convention, not a syntax rule of C++.

The last thing that you should note is that we can use more than one variable in an expression, i.e. `momsAge - yourAge`.

Now you should be able to complete the main activity.

Activity solution

```
//Calculates and displays the quotient of two numbers
#include <iostream>
using namespace std;

int main( )
{
  int m, n;
  cout << "Enter two numbers: ";
  cin >> m >> n;
  cout << "The quotient of " << m << " and " << n << " is " << m / n << endl;
  return 0;
}
```

Discussion

Note the variable names in this program. You might think that `m` and `n` are not very descriptive names. Perhaps you think that names like `numerator` and `denominator` would have been better. Well in this

case it is doubtful whether fancier names would contribute to the readability of the program. There is another convention that we use when declaring variables: When we declare a variable that is only meant to store a value, and the value doesn't represent anything specific (like an age, an amount of money or a number of people) then we normally use a single lowercase letter of the alphabet like `m` or `n`, or `i`, `j` or `k`. This is because a longer name doesn't add any information, and can in fact clutter the code and make it less readable.

Activity 3.b

Change the program of the previous activity to display the following on the screen

```
Enter two numbers: 23 5
The remainder of 23 divided by 5 is 3
```

In other words, instead of calculating and displaying the quotient, the program should calculate and display the remainder.

Test yourself

We have provided two solutions to this problem. The first is given in the solution to the first subactivity below, and the other is given in the activity solution further on. You ought to be able to produce the first solution with the knowledge that you have so far. If you are unable to produce the first solution yourself, you need to go back to the previous lesson (especially Exercise 2.3) and the first activity of this lesson. Make sure that you understand everything completely. Then attempt this problem again.

When you have written a program, compare it to the one below. Then work through the subactivity before simplifying your program and comparing it to the final activity solution.

Subactivity 3.b.i

Write a program for the main activity using the solution to Exercise 2.3.

Subactivity solution

```
//Calculates the remainder of one number divided by another
#include <iostream>
using namespace std;

int main( )
{
  int m, n;
  cout << "Enter two numbers: ";
  cin >> m >> n;
  cout << "The remainder of " << m << " divided by " << n;
```

```
  cout << " is " << m - m / n * n << endl;
  return 0;
}
```

Note that we displayed the message and the expression in separate `cout` statements simply because the line became very long. It would have been quite acceptable to place them all in a single `cout` statement.

Subactivity 3.b.ii

Type in the following program:

```
//Inputs a time and a number of hours
//Calculates new time after number of hours has elapsed
#include <iostream>
using namespace std;

int main( )
{
  int time, hours;
  cout << "Enter the time now (only the hours 0 ... 23): ";
  cin >> time;
  cout << "Enter a number of hours: ";
  cin >> hours;
  cout << "In " << hours << " hours the time will be ";
  cout << (time + hours) % 24 << endl;
  return 0;
}
```

In your study notebook, draw a table with four columns and at least ten or more rows. Label the columns *time*, *hours*, *predicted* and *actual*.

Now compile and run the program. Enter two numbers on the keyboard, and write them down in the first two columns of the first row of the table. Before pressing <Enter> after the second number, write down what you predict will be the output of the expression in the third column. Then press <Enter> and write down what the computer actually outputs in the fourth column.

Repeat this process a number of times until you can successfully predict what the output will be for any input, and have worked out (in general) what the `%` operator does.

If you are not too sure which numbers to test, take a peek at the subactivity solution below.

Subactivity solution

We tried the following values for time and hours:

	time	*hours*	*predicted*	*actual*
1.	*0*	*0*		
2.	*0*	*13*		
3.	*0*	*55*		
4.	*0*	*24*		
5.	*23*	*1*		
6.	*23*	*5*		
7.	*12*	*24*		
8.	*12*	*48*		
9.	*12*	*73*		
10.	*144*	*144*		

The interesting results for this choice of input values were as follows:

- Row *3*. A rather strange value was displayed.
- Row *4*. This indicates that a "wrap-around" is occurring. 24 hours after midnight (0) it is again midnight.
- Rows *5* and *6*. Tests whether the idea of the wrap-around is correct.
- Rows *7*, *8* and *9*. Tests whether the wrap-around works for multiple days.
- Row *10*. We used an illegal input for the first number to test whether the `%` operator works for any numbers. We know that 144 is a multiple of 24, as is 288.

Hopefully you discovered in this exercise that the `%` operator gives the remainder. In other words, if we have two numbers `i` and `j`, the expression `i % j` gives the remainder of `i` divided by `j`.

Note that every value that is output by the program, namely the value of the expression, is always a number from 0 to 23. This is because whatever the value of `(time + hours)` is, when it is divided by 24, the only possible remainders are 0 up to 23.

Discussion

The `%` operator is very useful in a number of interesting situations, so it is worthwhile spending some time making sure you understand what it does.

As stated above, it gives the remainder of one number divided by another. Remember that this is with respect to *integer division*. Consider the following examples:

5 divided by 3 is 1 remainder 2	so `5 / 3` is 1	and `5 % 3` is 2.
4 divided by 2 is 2 remainder 0	so `4 / 2` is 2	and `4 % 2` is 0.
3 divided by 6 is 0 remainder 3	so `3 / 6` is 0	and `3 % 6` is 3.
23 divided by 4 is 5 remainder 3	so `23 / 4` is 5	and `23 % 4` is 3.

Activity solution

The second solution is:

```
//Calculates the remainder of one number divided by another
#include <iostream>
using namespace std;

int main( )
{
  int m, n;
  cout << "Enter two numbers: ";
  cin >> m >> n;
  cout << "The remainder of " << m << " divided by " << n;
  cout << " is " << m % n << endl;
  return 0;
}
```

Important points in this lesson

Programming concepts

In this lesson we have seen what a variable is, namely a memory position where a value is stored. A variable has a name so that we can refer to the memory position. To create a variable and give it a name, we use a declaration statement. The general format of a declaration statement is

```
int VariableName;
```

where *VariableName* is any sequence of alpha-numeric characters, or the underscore character, but that does not start with a numeric character.

We can declare more than one variable in a declaration statement by listing them (in the place of *VariableName*) and separating their names by commas.

We also learnt how to use a `cin` statement to input a value from the keyboard and store it in a variable. The general format of an input statement is

```
cin >> VariableName;
```

We can input more that one value (i.e. get values for more than one variable) using a single `cin` statement. This is done by listing the variable names and separating them by the `>>` operator.

We saw that the `>>` operator is used with `cin` whereas `<<` is used with `cout`. The `cout` statement can be used to output a mixture of string messages, variable values and values of expressions. Expressions can contain one or more variables in the place of integer literals.

Finally we saw that the `%` operator can be used to calculate the remainder of one number divided by another. The remainder is always a number from 0 up to 1 less than the divisor. For example, `n % 10` will always give a remainder of 0 up to 9, no matter what the value of `n` is.

To sum up: If you think back to the end of Lesson 1, you will remember that we said that the general structure of a program is:

```
//DescriptiveComment
#include <StandardHeaderFile>
using namespace std;

int main( )
{
   StatementSequence;
}
```

In this lesson we have seen that there are other statements that can be included in *StatementSequence*, namely declarations and `cin` statements.

Programming principles

In general, we place declarations at the beginning of *StatementSequence*, although we can put declarations anywhere in the program. We must declare a variable before we can refer to it however, and so it is best to declare all variables at the beginning of the program.

Another aspect of good programming style is to use meaningful and descriptive variable names. We generally use lowercase letters for a variable name. However, if the name consists of more than one word, we use uppercase for the first letter of each subsequent word and join all the words together. However, if a variable is only used to store a number that doesn't represent some other amount, we prefer to use a single letter of the alphabet, normally `m` or `n`, or `i`, `j` or `k`.

Exercises

Exercise 3.1

Write a program that inputs three values and displays them on a single line in reverse order.

Exercise 3.2

Look at the following program and then answer the questions that follow *without* typing in the program and testing it:

```
#include <iostream>
using namespace std;

int main( )
{
  int x, y, z;
  cout << "Enter values for variables x, y and z:" << endl;
  cin >> x >> y >> z;
  cout << "x + y / z is " << x + y / z << endl;
  cout << "x % z is " << x % z << endl;
  cout << "y * z / x + 2 is " << y * z / x + 2 << endl;
  return 0;
}
```

(i) What will the output be if the user enters 2, 6 and 4?

(ii) What will the output be if the user enters 5, 1, 3?

(iii) If the last output statement is changed to

```
cout << "y * (z / x + 2) is " << y * (z / x + 2) << endl;
```

what will the output of this statement be if the user enters the same values as specified in (i) and (ii)?

Exercise 3.3

Lorraine inherited her grandmother's old cookbook, but all the oven temperatures are given in Fahrenheit, and her oven is only calibrated in Celsius. Write a program to help her. The formula for converting from Fahrenheit to Celsius is

$$C = 5.(F - 32)/9$$

Lesson 4

Assignment statements

Purpose of this lesson

In the previous two lessons, we wrote programs that displayed the results of calculations. We used variables to enable our programs to input values from the keyboard and work with those values instead of just doing calculations on literal values. In this lesson, we see how we can store the result of a calculation in a variable. This is done with a so-called assignment statement.

Activity 4.a

Write a program to add two periods of time (e.g. lengths of tracks on a CD). Each time must be input as a number of minutes and seconds. The output must be the sum of the two input times, and must also be in the form of minutes and seconds.

Test yourself

You should be able to write a program for this problem with the knowledge you have so far. Think about it. You can input four integers: minutes and seconds for the first period of time and minutes and seconds for the second. To add them together, add the minutes and add the seconds separately. The only complication is if the total number of seconds is greater than 60, in which case the total number of minutes needs to be incremented by 1, and the total number of seconds decremented by 60. This sounds complicated, but it can be done quite easily with the `/` and `%` operators.

You should be aware however, that things are going to get quite messy. The expressions for the total number of minutes and seconds are going to be long and complicated. What we need is a way to store the result of intermediate calculations so that we can use them in further expressions to make them simpler.

Subactivity 4.a.i

Write a program for the main activity, but ignore the problem of the seconds adding up to more than 60. In other words the program can give the following output:

```
(Specify only minutes and seconds, separated by spaces)
Enter a period of time: 1 35
Enter another one: 2 27
The total time is 3 minutes and 62 seconds
```

Subactivity solution

```
1  //Adds two times (specified as minutes and seconds) FIRST ATTEMPT
2  #include <iostream>
3  using namespace std;
4
5  int main( )
6  {
7     int mins1, mins2, secs1, secs2;
8
9     cout << "(Specify only minutes and seconds, separated by spaces)" << endl;
10    cout << "Enter a period of time: ";
11    cin >> mins1 >> secs1;
12    cout << "Enter another one: ";
13    cin >> mins2 >> secs2;
14    cout << "The total time is " << mins1 + mins2 << " minutes";
15    cout << " and " << secs1 + secs2 << " seconds" << endl;
16
17    return 0;
18 }
```

This is the first time we have used numeric characters (digits) in the names of variables. Variables `mins1` and `mins2` store the same kind of values, so we use the same name, but with a digit attached to distinguish them from one another. The same goes for `secs1` and `secs2`.

Subactivity 4.a.ii

Once you have got the above program to work correctly, edit it as follows:

- Declare an additional variable called `totalMins`. Add it to the declaration in line 7.
- Replace the expression `mins1 + mins2` in line 14 with `totalMins`.
- Insert the assignment statement `totalMins = mins1 + mins2;` between lines 13 and 14.

Compile and run the program again. It should do exactly the same thing as the first version.

Discussion

An *assignment statement* is used to store the result of a calculation in a variable. In this case, the value of the expression `mins1 + mins2` is calculated and assigned to (i.e. stored in) variable `totalMins`. The equals sign is called the *assignment operator*. We place a variable on the left-hand side of the assignment operator and an expression on the right-hand side. Note that we cannot place an expression on the left-hand side because the computer cannot store a value in an expression.

We can indicate the change that the assignment statement causes on the values of the variables by two sets of variable diagrams:

	mins1	*mins2*	*secs1*	*secs2*	*totalMins*
Line 13:	*1*	*2*	*35*	*27*	

After the assignment statement, the situation will change as follows:

	mins1	*mins2*	*secs1*	*secs2*	*totalMins*
Line 14:	*1*	*2*	*35*	*27*	*3*

Subactivity 4.a.iii

Do the same for the remaining expression in the last output statement. Compile and run your program to test whether it still does the same thing as the original program.

Subactivity solution

Your program should now look like this:

```
//Adds two times (specified as minutes and seconds) SECOND ATTEMPT
#include <iostream>
using namespace std;

int main( )
{
   int mins1, mins2, secs1, secs2, totalMins, totalSecs;

   cout << "(Specify only minutes and seconds, separated by spaces)" << endl;
   cout << "Enter a period of time: ";
   cin >> mins1 >> secs;
   cout << "Enter another one: ";
   cin >> mins2 >> secs2;
   totalMins = mins1 + mins2;
   totalSecs = secs1 + secs2;
   cout << "The total time is " << totalMins << " minutes";
   cout << " and " << totalSecs << " seconds" << endl;

   return 0;
}
```

Discussion

In these subactivities we have seen that we can always introduce a variable to a program without changing the program's functionality. We often do this to store the value of a calculation for use later in the program.

In fact, it is better not to calculate the value of an expression in an output statement as we have been doing up to now. It is better to separate processing (i.e. calculating the values of expressions) from inputting and outputting. To separate processing from outputting, it is often necessary to introduce one or more additional variables.

Subactivity 4.a.iv

Now write a program to input a large number of seconds (it could be 60 or more) and to display the time as a number of minutes and seconds. Hint: Use `/` and `%`.

Subactivity solution

```
//Convert a large number of seconds to minutes and seconds
#include <iostream>
using namespace std;

int main( )
{
   int secs, totalMins, totalSecs;

   cout << "Enter a large number of seconds: ";
   cin >> secs;
   totalMins = secs / 60;
   totalSecs = secs % 60;
   cout << "This is " << totalMins << " minutes";
   cout << " and " << totalSecs << " seconds" << endl;

   return 0;
}
```

If you only used one variable in your program and included expressions in your output statements instead of introducing new variables, that was naughty of you. From now on, remember to introduce variables to avoid having to include expressions in output statements.

You should now be able to write the program for the main activity by combining the ideas from the above subactivities.

Activity solution

The final program looks like this:

```
//Adds two times (specified as minutes and seconds) FINAL VERSION
#include <iostream>
using namespace std;

int main( )
{
   int mins1, mins2, secs1, secs2, totalMins, totalSecs;

   cout << "(Specify only minutes and seconds, separated by spaces)" << endl;
   cout << "Enter a period of time: ";
   cin >> mins1 >> secs1;
   cout << "Enter another one: ";
   cin >> mins2 >> secs2;
```

```
   totalMins = mins1 + mins2 + (secs1 + secs2) / 60;
   totalSecs = (secs1 + sec2) % 60;
   cout << "The total time is " << totalMins << " minutes";
   cout << " and " << totalSecs << " seconds" << endl;

   return 0;
}
```

Discussion

You might well have introduced another variable for this program, say `manySecs`, and used it to store the value of `secs1 + secs2`. This would save having to calculate `secs1 + secs2` twice.

So this is another reason for introducing an additional variable to a program: to store the value of a calculation to save having to perform the calculation more than once.

As you can see, one can't make hard and fast rules about what variables to introduce. All that you can do is to apply the principle of making your code as readable / understandable and efficient as possible.

Activity 4.b

There are a number of other assignment operators in C++, namely `+=`, `-=`, `/=` and `*=`, as well as `++` and `--`. Explain what each of them does.

Test yourself

To investigate these operators, write a short program that uses one (or more) of them. Note that they are assignment operators. This should give you a clue as to how to use them.

The first subactivity is an example of using the `+=` operator.

The next two subactivities are examples of using the `++` operator, which (like `--`) works differently.

Subactivity 4.b.i

Write a program that first inputs a number from the keyboard into a variable called `n`, and then has the assignment statement `n += 27;`

The program should then output the value of `n`.

Subactivity solution

```
//Tests compound assignment statements
#include <iostream>
using namespace std;

int main( )
```

```
{
   int n;
   cout << "Enter a number: ";
   cin >> n;
   n += 27;
   cout << "Now the value of the number is " << n << endl;

   return 0;
}
```

Discussion

From the output of this program you should be able to see that the `+=` operator adds the value on the right-hand side to the variable on the left-hand side.

Note that the value on the right-hand side of `+=` can be a literal, a variable or a more complicated expression.

Subactivity 4.b.ii

Edit the program you wrote for the previous subactivity and replace the assignment statement with the statement

```
   n++;
```

Before you compile and run the program, try to predict the output.

Discussion

The `++` operator simply increments a variable by 1.

Note that the `++` operator can only be applied to a variable, and not to a literal value or an expression.

As you can see, the `++` operator is a unary operator in that it only operates on a single variable. All the other operators that we have seen so far are binary operators because they are always used with (i.e. between) two values. You do know of one other unary operator, namely unary `-`. (The expression `-i` gives the value stored in `i` with a changed sign. In other words, if `i` is positive, `-i` is negative and if `i` is negative, `-i` is positive.)

Note that unary minus is only used in prefix form. In other words, it is always placed before the value it operates on. We used the postfix form of `++` above. In other words, we placed it after the variable it operates on. However, `++` can also be used in prefix form. Try it now. Edit the program and replace `n++;` with `++n;`.

The pre- and postfix forms of the `++` operator appear to do the same thing. To see the difference, do the next subactivity.

Subactivity 4.b.iii

Predict what the output of the following program will be:

```
//Tests unary increment operator
#include <iostream>
using namespace std;

int main( )
{
   int m, n;
   cout << "Enter a number: ";
   cin >> n;
   m = n++;
   cout << "m is " << m << " and n is " << n << endl;
   return 0;
}
```

Now replace `n++` with `++n` and predict the output of program.

Subactivity solution

Compile and run the program to check whether your predictions were correct.

Discussion

Before we discuss the difference between the prefix and postfix forms of `++`, note how we use `n++` as an expression rather than as a statement on its own (as we did in the previous subactivity). We can do this because, apart from incrementing a variable, the operation returns a value. In other words, the variable together with the operator can act as an expression.

The value that it returns depends on where the operator is placed. If it is used in its postfix form, the operation returns the value of the variable before incrementing it. If it is used in its prefix form, it returns the value of the variable after incrementing it.

Note that the `++` operator can be used in the middle of a long and complicated expression, e.g.

```
i = j * (++k) -23;
```

Activity solution

The operators `+=`, `-=`, `*=` and `/=` are all short-hand versions of assignment statements that are used to change the value of a variable. The statement

`Variable += Expression;`

adds the value of *`Expression`* to *`Variable`*. Similarly

```
Variable -= Expression;
Variable *= Expression;
Variable /= Expression;
```

subtracts *Expression* from *Variable*, multiplies *Variable* by *Expression*, and divides *Variable* by *Expression* (respectively).

The unary operators ++ and -- are used to increment and decrement variables respectively. Both can be used in prefix or postfix form, i.e. before or after the variable that they change. The pre- and postfix forms have the same effect on the variable; the difference lies in the value that they return when used in an expression (or as an expression). The prefix form returns the value of the variable after it has been changed, whereas the postfix form returns its value before it was changed.

Important points in this lesson

Programming concepts

We can change the value of a variable in two ways: Firstly (as we saw in the previous lesson) we can input a value into a variable using a `cin` statement. Secondly (as we saw in this lesson) we can assign a value to a variable using an *assignment statement*.

The general format of an assignment statement is

```
Variable = Expression;
```

The value of *Expression* is calculated first, and then the result is stored in *Variable*. *Expression* may therefore contain (one or more occurrences of) *Variable*. In such a case, the old value of *Variable* is used to calculate *Expression*, and the result is stored back in *Variable*.

Then we learnt about the so-called *compound assignment operators*, namely +=, -=, *= and /=. They provide a short-hand notation for assignment statements for changing the value of a variable. For example

```
Variable -= Expression;
```

subtracts the value of *Expression* from *Variable*.

Finally we saw that the two unary operators ++ and -- add 1 and subtract 1 from the variable they are applied to, respectively. ++ is called the *unary increment operator* and -- is called the *unary decrement operator*. We saw that these operators, together with the variable they are operating on, return a value. They can therefore be used where an expression is required or in a longer complicated expression. The value returned by such an expression depends on whether the operator is used before the variable (prefix form) or after it (postfix form). If the operator is used before it, the returned value is the final (changed) value of the variable, whereas if it is used after it, the returned value is the original value of the variable (i.e. before it was changed).

Programming principles

We saw the following tips and tricks to make programs more readable and less prone to errors:

- Use numerical digits in variable names to distinguish different variables that are used to store the same kinds of values.
- Remember to initialise variables (either by assigning a value to them or by inputting a value for them) before using them.

- Rather declare additional variables than do calculations in an output statement.
- If you find you are doing exactly the same calculation over and over, declare a variable and store the result of the calculation (once) in it.

Exercises

Exercise 4.1

Redo Exercise 3.3, but use a variable to store the result of the calculation in instead of doing the calculation in the output statement.

Exercise 4.2

Sam has a small business which does packaging and posting. Clients come to him with a number of items (of the same size and shape), and he has to pack them into boxes before posting them to some customer of theirs. He works as follows: He packs as many of the items in a box as he can, and then calculates how many boxes he will need, and how many items will be left over (i.e. that won't fit into a box).

Write a program to input a number of items, and a number of items that fit in a box. The program should calculate how many boxes will be needed, and how many items will be left over.

Exercise 4.3

What will the value of variable `n` be after the following series of statements?

```
int n = 10;
n += 3;
n /= 2;
n++;
n %= 4;
n -= 5;
```

Lesson 5

Variable diagrams

Purpose of this lesson

When a computer is executing a program, you cannot see what the computer is actually doing inside. How does it actually do what the program tells it to do? What happens as it executes each program statement? We need to find ways to understand what goes on inside that box we call a computer, and variable diagrams are a useful way to do that.

Activity 5.a

We have seen variable diagrams in previous lessons. We give a short program below. First, work out what it does. Then draw a series of variable diagrams to show the successive values of `thisOne`, `thatOne`, `thisOne2`, and `theOther` after each of lines 10, 11 and 12. Finally, state what the output of the program will be.

```
1  //Program to illustrate arithmetic operators
2  #include <iostream>
3  using namespace std;
4
5  int main( )
6  {
7     int thisOne, thatOne, thisOne2;
8     const int theOther = 5;
9
10    thisOne = 3;
11    thatOne = 2 * thisOne + 7 / thisOne * theOther;
12    thisOne2 = thatOne * thisOne / 9 + 8;
13
14    cout << "thisOne = " << thisOne;
15    cout << "; thatOne = " << thatOne << endl;
16    cout << "thisOne2 = " << thisOne2;
17    cout << "; theOther = " << theOther << endl;
18
19    return 0;
20 }
```

Test yourself

If you understood the simple variable diagrams that we drew in previous lessons, you ought to be able to complete this activity quite easily.

Normally, if you manage to work out the solution to the main activity yourself, we suggest that you skip forward to the end of the lesson. However, we suggest that you now work through the discussion that follows. You will notice that we do not use the usual subactivity format for this activity, and that we tend to tell you things more directly than usual.

Activity solution

In the problem statement above, we suggested that you draw variable diagrams for lines 8 to 10, but to clarify the entire program, we will discuss it line by line from the beginning.

Line 1: `//Program to illustrate arithmetic operators`

The compiler ignores everything after `//` since it is a comment. Comments are included so that the person reading the program can understand more easily what it does.

Line 2: `#include <iostream>`

This line tells the compiler to include the names defined in the `iostream` header file. Without this, the compiler will not recognise `cout` and `endl`.

Line 3: `using namespace std;`

This line tells the compiler to use the namespace called `std`. It allows the compiler to recognise the names defined in standard header files like `iostream`. Without this statement, we should prefix each of the names with `std::`.

Note that line 4 is left blank to make the program easier to read for the programmer or for whoever else needs to read the program.

Line 5: `int main( )`

This indicates the beginning of the main function of the program. Every C++ program must have a function called `main`. The reserved word `int` indicates that it will return an integer value to the operating system when the program terminates. This matches the statement `return 0;` in line 19.

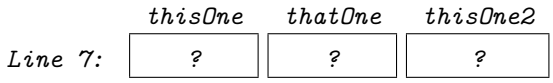
Line 6: {

The open curly bracket (or brace) indicates the beginning of the body of the main function.

Line 7: `int thisOne, thatOne, thisOne2;`

This line declares three integer variables called `thisOne`, `thatOne`, and `thisOne2`. From this declaration statement the compiler knows that it must set aside a separate memory position that can hold an integer for each of these three variables.

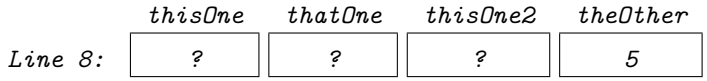
None of these variables have a value yet. This does not mean that they are set to zero, but that their value is completely unknown or undefined. We can already draw diagrams for these variables, and indicate these unknown values with `?`.



Line 8: `const int theOther = 5;`

This line declares a constant called `theOther`, and gives it a value of 5. The reserved word `const` means that its value cannot be changed at any point during the program.

We can draw the following variable diagrams for it:



Discussion

There is an important difference between declaring a variable (as in line 7) and declaring a constant (line 8). When declaring a constant, we use a statement of the form

```
const int ConstantName = Value;
```

The `const` reserved word tells the compiler that the integer constant called *`ConstantName`* will have the value of *`Value`* throughout the program, and that its value cannot change while the program is running. In fact, we generally use uppercase letters for constant names to distinguish them from variables. With this convention, the constant declaration would be

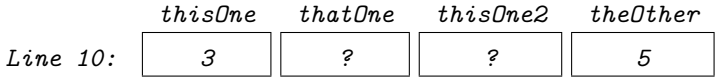
```
const int THE_OTHER = 5;
```

Note how we use the underscore character to separate multiple words in the name of a constant.

Line 10: `thisOne = 3;`

This line assigns the value 3 to the variable `thisOne`.

Variables `thatOne` and `thisOne2` will be given values later on in the program, and at the moment are still undefined. The constant `theOther` keeps the same value throughout the program. The variable diagram therefore changes to



Line 11: `thatOne = 2 * thisOne + 7 / thisOne * theOther;`

This is a more complicated assignment statement than the one in the previous line. We need to do a calculation to work out the value that will be assigned to the variable `thatOne`. To evaluate this line, we need to know the values of `thisOne` and `theOther`. We can get both of these values from the variable diagram that we drew for line 8.

In working out this value, we must also take care to follow the correct operator precedence (namely * and / before either + or -) and of the left-to-right rule for operators of equal precedence:

```
  2 * 3 + 7 / 3 * 5
    |       |
=   6   +   2   * 5
                |
=   6   +       10
        |
=       16
```

Notice that we have done the * and / operations before +, and that, where / came before *, we also used the left-to-right rule. This is termed *operator precedence*. If you feel unsure about this, look back at Lesson 2.

The next set of variable diagrams is therefore:

	thisOne	*thatOne*	*thisOne2*	*theOther*
Line 11:	*3*	*16*	*?*	*5*

Line 12: `thisOne2 = thatOne * thisOne / 9 + 8;`

Using the values in the diagrams for line 11, work out what value this statement will store in variable `thisOne2` and then compare it with the answer we give below. Note how the operators with the same precedence (* and /) are evaluated from left to right.

```
  16 * 3 / 9 + 8
     |
=   48   / 9 + 8
         |
=        5   + 8
             |
=            13
```

This gives us the next set of variable diagrams:

	thisOne	*thatOne*	*thisOne2*	*theOther*
Line 12:	*3*	*16*	*13*	*5*

The remaining lines of the program do not change the values of any variables, and so we do not need to draw any more variable diagrams.

You should now have a clear idea of variable diagrams and how to draw them yourself. In later lessons, we will use variable diagrams again, but with more complicated problems.

We have finished with the variable diagrams, but we have not quite finished with the program. What about lines 14 to 17? What do they do? We repeat them here:

```
cout << "thisOne = " << thisOne;
cout << "; thatOne = " << thatOne << endl;
cout << "thisOne2 = " << thisOne2;
cout << "; theOther = " << theOther << endl;
```

Of course, these lines do not change the values of any variables. As you will recognise from a previous lesson, the `cout` instruction displays certain values on the screen. Working from the final variable diagrams we have just completed, the computer will display:

```
thisOne = 3; thatOne = 16
thisOne2 = 13; theOther = 5
```

Activity 5.b

Type in the program below (without the line numbers) and test it out. Hint: Press <Enter> every time the program displays an instruction to see the next instruction.

```
1  //Think of a number
2  #include <iostream>
3  using namespace std;
4
5  int main( )
6  {
7     int number = 40;
8     int answer;
9     cout << "Think of a number between 30 and 50. Write it down" << endl;
10    cout << "Then do the following calculations on paper:" << endl << endl;
11    cin.get( );
12    cout << "Double it" << endl;
13    cin.get( );
14    answer = number * 2;
15    cout << "Add 29 to this" << endl;
16    cin.get( );
17    answer = answer + 29;
18    cout << "Double the result again" << endl;
19    cin.get( );
20    answer = answer * 2;
21    cout << "Subtract the original number from your answer" << endl;
22    cin.get( );
23    answer = answer - number;
24    cout << "Divide the answer by your original number and throw away any remainder" << endl;
25    cin.get( );
26    answer = answer / number;
27    cout << "Your final answer is " << answer << endl;
28    return 0;
29 }
```

Draw variable diagrams to show the declaration of variables and to show how their values change.

Test yourself

You should understand many of the lines of this program, as you have seen many of the statements before. There are a few lines that may seem rather strange. You should be able to get a good idea of what they do just by running the program. We give a number of subactivities that give some further information on the new concepts used in this program.

Subactivity 5.b.i

Predict the output of the following program:

```
//What's the output?
#include <iostream>
using namespace std;

int main( )
{
   int i = 13;
   cout << i << endl;
   return 0;
}
```

Say the declaration of variable `i` is changed to `int i;` Predict the output once again.

Subactivity solution

If you are not sure of your predictions, you had better type in, compile and run the above program - because we aren't going to tell you what the output is!

Discussion

There are some interesting things to note about this program, however:

The statement `int i = 13;` declares variable `i` and initialises it to 13 in one step.

If we replace the declaration statement with `int i;` we have no idea what value `i` has. Remember that a declaration statement just sets a memory position aside for an integer, but that memory position can have some arbitrary value stored in it. If the program outputs the value of an uninitialised variable, the output is unpredictable. We generally use a question mark to indicate an uninitialised value in a variable diagram.

Have you worked out what `cin.get( );` does yet? Here is a subactivity to confuse you if you think you know.

Subactivity 5.b.ii

Run the program of the main activity again and this time type `ABC` before pressing <Enter> the first time.

Describe what happens.

Subactivity solution

This time the program does not wait for the user to press <Enter> between each instruction, but displays the next three instructions immediately before giving the user a chance to press <Enter> again.

Subactivity 5.b.iii

Write a program to input a number from the keyboard. The program must then multiply the number by 2 and display the result. There are two restrictions on your program, however:

(i) The program may only use one variable.

(ii) The program may not include an expression (i.e. with calculations) in a `cout` statement.

Subactivity solution

```
//Doubles a number and displays the result
#include <iostream>
using namespace std;

int main( )
{
   int n;
   cout << "Enter a number: ";
   cin >> n;
   n = n * 2;
   cout << "Doubled: " << n << endl;
   return 0;
}
```

Discussion

Here we see an interesting thing. We are assigning a value to a variable which also appears in the expression on the right-hand side of the assignment operator. For example, the assignment statement

```
n = n + 5;
```

looks wrong. How can `n` equal `n` plus 5? It seems strange because it looks like a mathematical equation. But you must remember that an assignment statement is not an equation. It is an instruction to store a value in a variable.

What happens is this: The computer first calculates the value of the expression on the right hand side (i.e. `n + 5`) using the original value of `n`. Then it assigns the result to the variable on the left-hand side

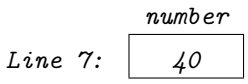
(which happens to be `n` as well). The fact that this is the same variable does not matter because of the order in which things are done, namely first evaluate the expression and then assign the value. So this statement will add 5 to whatever is stored in variable `n` and store it back in variable `n`.

Activity solution

The first statement in the program that will memory allocation (or change the value of variables) is line 7:

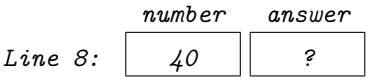
Line 7: `int number = 40;`

This declares a variable called `number` and initialises it to the value 40. We indicate this by means of a variable diagram as follows:



Line 8: `int answer;`

This declares variable `answer`.



As stated above, we indicate that variable `answer` is uninitialised (i.e. it does not have a value yet) by means of a question mark. It is unnecessary to initialise `answer` at this point because its value will be changed by the assignment statements later on.

Line 9, 10, 12, 15, 18, 21, 24 and 27: `cout << "..." << endl;`

These are output statements like we have seen before. They have no effect on any of the variables. Line 27 outputs the final value of variable `answer`.

Line 11, 13, 16, 19, 22 and 25: `cin.get( );`

These statements appear to get the program to wait for the user to press <Enter>. However, as Subactivity 5.b.ii shows, it is more accurate to say that `cin.get( );` removes one keystroke from the input stream `cin`. These statements have no effect on the values of the variables.

Line 14, 17, 20, 23 and 26: `answer = ...;`

These are assignment statements to calculate new values for variable `answer`. Lines 17, 20, 23 and 26 are interesting because variable `answer` appears both on the left hand side of `=` and in the expression on the right hand side of `=`. These lines have the effect of updating the value of variable `answer`. (We could have declared four additional variables to store the values of each calculation, but this is not necessary. Using the same variable over and over saves us having to do so.)

We can indicate the change brought about by each assignment statement by means of a series of variable diagrams:

	number	answer
Line 14:	40	80

	number	answer
Line 17:	40	109

	number	answer
Line 20:	40	218

	number	answer
Line 23:	40	178

	number	answer
Line 26:	40	4

Try changing the initial value of variable `number` to any number between 30 and 50 and you will see that the output of the program remains the same. This shows that the calculations all give the same result no matter what number the user thinks of.

Discussion

A series of variable diagrams like those above are like a series of snap shots of the computer's memory. The "photos" are "taken" just after the program has executed the line as indicated.

Drawing all these diagrams can be quite a bother since there is so much repetition. It would be easier to stand in front of a white- (or black-) board and just draw one set of boxes. Then, as you step through the program, erase the values that change and write in the new values. You could still write the line number in front to help you keep track of where you are.

Important points in this lesson

Programming concepts

The purpose of this lesson was to show how variable diagrams can be used to work out what programs do, particularly as they get more difficult. Variable diagrams can also be a very good way of tracking down bugs which are difficult to find. We will come back to both of these aspects in future lessons.

In addition to learning about variable diagrams, this lesson introduced the following concepts:

- We can initialise a variable while declaring it. This can be done in the format

  ```
  int Variable = Value;
  ```

- We can declare constants to improve the readability and maintainability of our code. The format for a constant declaration is

  ```
  const int ConstantName = Value;
  ```

- When we declare a variable, we are telling the computer to set aside a memory position to store a value that will probably change during execution of the program. By contrast, when we declare a constant, we give it a value that will remain unaltered throughout the program. When we declare a variable we *may* provide a value to initialise it, but when we declare a constant, we *must* provide a value.
- In many computer languages, a variable only has a value once it has been initialised (i.e. once the program actually assigns it a value). Before initialisation, the value of a variable is undefined and that is why we use a question mark in its variable diagram to indicate that its value is unknown (and probably garbage).

Programming principles

- Remember to initialise variables.
- Declare constants whenever you can.
- Use uppercase letters for the names of constants (to distinguish them from variables). If the name consists of more than one word, use the underscore character to separate the words, eg.

```
const int DAYS_OF_YEAR = 365;
```

- If it becomes difficult to work out what a program does, draw variable diagrams!

  We indicate the value of an uninitialised variable with a question mark to show that the value is unknown. The following statements in a program have an effect on the variable diagrams we draw:

  – Declaration statements: We draw a box for each variable, and label them with their names.
  – Input statements: We write the values input from the keyboard in the appropriate boxes.
  – Assignment statements: We change the value in the box of the variable (on the left-hand side) to the value determined by the expression (on the right-hand side).

Exercises

Exercise 5.1

Consider the program below and draw variable diagrams to show the value of each variable for each statement, and give the exact output of the program.

```
1  //Tracing practice
2  #include <iostream>
3  using namespace std;
4
5  int main( )
6  {
7     int j, k, m, n;
8     j = 3;
9     k = 2;
10    m = (j * j - 6) / k;
11    n = j * j - 6 / k;
12    cout << j << " " << k << " " << m << " " << n << endl;
13    j = m / 2 + 3 * j;
```

```
14      k = j - m * n;
15      cout << j << " " << k << " " << m << " " << n << endl;
16
17      return 0;
18  }
```

Exercise 5.2

Rewrite the solution to Activity 5.b using a constant (instead of variable `number`), and using compound assignment statements (as discussed in Activity 4.b) in place of all the standard assignment statements. Then draw a series of variable diagrams to indicate that your program is equivalent to the original version.

Lesson 6

Floating point numbers

Purpose of this lesson

In the lessons that we have done so far, we wrote programs to display strings of characters and numbers to the screen. We have seen how to use integer variables to input, calculate and store numbers.

As far as we are concerned in this module, there are two main kinds of numbers. There are integers; we use them when we do not need to talk about fractions. Often, though, we need to talk about fractions as well, and then we use floating point numbers. For example, 2.3 metres or R 6.99 (or dollars, or whatever currency you are using). We use a decimal point to show this.

Activity 6.a

Write a program that inputs (from the keyboard) the prices of three different products, adds these together, and then calculates and adds 14% value added tax (VAT). Finally it displays the heading `The total price is`, followed by the total price with VAT included.

Test yourself

The most obvious thing you need to know to write this program is how to declare a variable to store a floating point number. The reserved word used for this purpose is `float`. In other words, to declare a floating point variable called `price`, use the statement

```
float price;
```

All the arithmetic and assignment operators for integers also work for floating point numbers (except the `%` operator for determining the remainder - this only works for integers).

If you do not know how to tackle this problem, continue with the sub-activities below where we teach you about the different things you need to know to write a program like this.

Subactivity 6.a.i

Write a program to input three floating point values and display their sum. Use a separate variable for the sum.

Subactivity solution

```
//Calculates the sum of three floating point values
#include <iostream>
using namespace std;

int main( )
{
   float value1, value2, value3, sum;

   cout << "Enter three values" << endl;
   cin >> value1 >> value2 >> value3;

   sum = value1 + value2 + value3;
   cout << endl << "The sum is " << sum << endl;

   return 0;
}
```

Note once again how we declare four floating point variables in a single declaration statement, input three floating point values in a single `cin` statement and calculate the sum in a separate assignment statement from the output statement.

Subactivity 6.a.ii

Now write a program that inputs (from the keyboard) the price of a product, calculates 14% VAT, and displays the message `VAT:` , followed by the VAT amount.

Subactivity solution

To write this program you need to know how to calculate 14% VAT on a given amount. That's easy! Just multiply the amount by 0.14. But how do we translate this into a C++ statement?

We need a variable name for the price the program should read. Call it `price`. Furthermore, we want to calculate a new value, the VAT on the price, for which we need another variable - say `vat`. Since we know that the VAT rate is 14%, we can represent it with a constant named `VAT_RATE`.

The complete program for this activity is given below.

```
//Calculates 14% VAT on a price
#include <iostream>
using namespace std;

int main( )
{
   const float VAT_RATE = 0.14;
   float price, vat;

   cout << "Enter the price ";
   cin >> price;

   vat = price * VAT_RATE;
```

```
   cout << endl << "VAT: " << vat << endl;

   return 0;
}
```

Now that you have completed the subactivities, you should be able to write the program for the main activity. Try to do it without looking at the solution below.

Activity solution

```
//Calculates the sum of three prices and adds 14% VAT
#include <iostream>
using namespace std;

int main( )
{
   const float VAT_RATE = 0.14;
   float price1, price2, price3, total, vat, vatIncl;

   //Input three prices
   cout << "Enter the prices of three items" << endl;
   cin >> price1 >> price2 >> price3;

   //Calculate total, VAT and total including VAT
   total = price1 + price2 + price3;
   vat = total * VAT_RATE;
   vatIncl = total + vat;
   cout << "The total price is " << vatIncl << endl;

   return 0;
}
```

Since there are usually several different ways to write a program, yours may differ slightly from this.

Note the use of comments in the code of our solution to explain what the two main parts of the code do. From now on, we will regularly use comments in the code (not just at the beginning of the program) to provide explanations of sections of code. You should do the same in your programs.

Remember to test your program carefully by first working out a few test cases on paper and then seeing whether your program gives the same answer. If your program does not give the correct answers, it might be useful to put in a few extra `cout` statements to display the intermediate values like `total` and `vat`. For example, you could modify the program we have just given as follows:

```
//Calculates the sum of three prices and adds 14% VAT
#include <iostream>
using namespace std;

int main( )
{
   const float VAT_RATE = 0.14;
```

```
   float price1, price2, price3, total, vat, vatIncl;

   //Input three prices
   cout << "Enter the prices of three items" << endl;
   cin >> price1 >> price2 >> price3;

   //Calculate total, VAT and total including VAT
   total = price1 + price2 + price3;

   cout << endl << "Total before VAT is " << total << endl;

   vat = total * VAT_RATE;
   cout << endl << "VAT on total is " << vat << endl;

   vatIncl = total + vat;
   cout << "The total price is " << vatIncl << endl;

   return 0;
}
```

Activity 6.b

Write a program to calculate the average of three marks. The marks must be input into three integer variables but the average that is output must be a floating point number.

Test yourself

The only complication of this program is the fact that the marks must be input into integer variables and the average calculated and displayed as a floating point number.

In fact, you can cheat in this program and declare all the variables as `float`s. This is the task of the first subactivity.

The second subactivity shows that if you change the type of the mark variables to `int`, the program gives the wrong answer. The final three subactivities show how to input into integer variables but do floating point calculations on them.

Subactivity 6.b.i

Write the program for the main activity, but use floating point variables throughout.

Subactivity solution

```
//Calculates the average of three marks
#include <iostream>
using namespace std;

int main( )
```

```
{
   float mark1, mark2, mark3, average;

   //Input the three marks
   cout << "Enter the values of three marks" << endl;
   cin >> mark1 >> mark2 >> mark3;

   //Calculate the average
   average = (mark1 + mark2 + mark3) / 3;
   cout << "The average is " << average << endl;

   return 0;
}
```

Easy-capeasy!

Discussion

Test your program out with a few different sets of marks. Note firstly that the user can enter all the numbers as integers. As far as the user is concerned, the program is storing the values as integers. But we know that the variables have been declared as `float`s.

Note secondly that the number of digits after the decimal point depends on the fraction calculated by the division operation. To see this, run the program three times, but each time increment the third number by 1. For example, enter 45 63 27, then 45 63 28 and lastly 45 63 29. If the average is an integer, it is output with no digits after the decimal point. If the average has a fractional part of one third, something like 45.3333 is displayed. If the fractional part is two thirds, something like 45.6667 is displayed. So if the fractional part is a recurring decimal, only a limited number of digits are displayed after the decimal point.

Subactivity 6.b.ii

Now change the declarations of the variables used to store the marks to `int`s and test the program again. Explain the output.

Subactivity solution

The program throws away the fraction part. The program now seems to be using a different version of the `/` operator, namely the one for integers.

Discussion

Remember how integer division in Lessons 2, 3, 4 and 5 threw away any fractional part? In fact, there are two versions of the `/` operator, one for integers and one for floating point numbers. The compiler decides which one is appropriate by looking at the operands (i.e. the values on either side of `/`). If they are integers, the compiler uses integer division. If they are floating point values, it uses floating point division.

If they are a mixture (one integer and one floating point value) it will use floating point division.

We want floating point division, so all we have to do is to ensure that (at least) one of the operands is a floating point value.

(Note that this doesn't only apply to division. It applies to addition, subtraction and multiplication as well.)

The next three subactivities give three ways in which we can get floating point division for this problem.

Subactivity 6.b.iii

Change the value by which you divide the sum of the three marks to 3.0. Recompile and run the program.

Subactivity solution

Now the second operand of `/` (namely 3.0) is a floating point value, so floating point division is used. Voila!

Subactivity 6.b.iv

Introduce a new variable called `sum` of type `float`. First calculate the `sum` and then divide it by 3 (not 3.0).

Subactivity solution

```
//Calculates the average of three marks
#include <iostream>
using namespace std;

int main( )
{
   int mark1, mark2, mark3;
   float sum, average;

   //Input the three marks
   cout << "Enter the values of three marks" << endl;
   cin >> mark1 >> mark2 >> mark3;

   //Calculate the sum and average
   sum = mark1 + mark2 + mark3;
   average = sum / 3;
   cout << "The average is " << average << endl;

   return 0;
}
```

This trick also solves the problem because the first operand of `/` is now a floating point value.

Discussion

There's something else funny going on here. Can you see it? We are adding three integer variables and storing the result in a floating point variable.

This is an example of something called an *implicit type conversion* where a value of one type (in this case `int`) is converted to a value of another type (in this case `float`). Implicit type conversion does not work the other way round however. In other words, you can't assign a floating point value to an integer variable. The compiler will complain (or at least give a warning). The reason why an implicit type conversion the other way round is not desirable is that we could lose information, particularly if a floating point number with a fractional part was converted implicitly to an integer (which doesn't have a fractional part).

Subactivity 6.b.v

Remove variable `sum` and this time replace the assignment statement with

```
average = float(mark1 + mark2 + mark3) / 3;
```

Recompile and run the program.

Discussion

This (third) method is called an *explicit type conversion*. We place the value to be converted in parentheses and precede it by the type that we want to convert to (in this case `float`).

In some situations it is preferable to use an explicit type conversion like this than to use one of the other two methods that we have seen. If we aren't dividing by another number, and if we don't want to introduce an additional variable, then an explicit type conversion like this is preferable.

In fact, we can get around the implicit type conversion problem explained in the previous **Discussion** section by doing an explicit type conversion from a floating point value to an integer (by placing the floating point value in parentheses and putting `int` in front). In this way we can tell the compiler to do the conversion no matter whether information is lost or not.

Activity solution

Any one of the three solutions developed in the last three subactivities is acceptable.

Activity 6.c

Edit the program you wrote for Activity 6.a so that the final total is output in the format

```
The total price is R XXXX.XX
```

Note that there must be precisely 2 digits after the decimal point, no matter whether there are cents in the final total or not.

Test yourself

As we saw in Subactivity 6.b.i, the computer decides how many decimal places to use when outputting a floating point number by looking at the fractional part of the number. If there is no fractional part, it displays the number without any digits after the decimal point. If there is a fractional part, it displays an "appropriate" number of digits after the decimal point. The first subactivity investigates what the "appropriate" number of digits is.

Sometimes we as programmers want to specify the format of output, however. In particular, we want to specify exactly how many places after the decimal point to display.

In the `iostream` header file, there are a number of member functions defined for manipulating `cout`. We can use them to display the output in the required format.

The rest of the subactivities show which member functions of `cout` are useful for our purpose.

Subactivity 6.c.i

Type in the following program.

```
//Illustrates the default precision
#include <iostream>
using namespace std;

int main( )
{
   float x;

   cout << "Enter a floating point number: ";
   cin >> x;

   cout << "The number is " << x << endl;

   return 0;
}
```

Compile and run this program and test it out with a number of values. First use small and large integer values. Then try floating point values with differing numbers of digits before and after the decimal point. Finally try a few negative values.

Describe in words (by writing down in your study notebook) what rules the computer uses to decide how many digits to display (before and) after the decimal point.

Subactivity solution

The following rules seem to be applied:

If the number of digits *before* the decimal point is more than six then the entire number is output in scientific notation. (We are not interested in scientific notation for the purposes of this study guide. Our efforts will therefore be to prevent floating point numbers from being displayed in this format.)

However, if the number of digits before the decimal point is six or less then the following happens:

If the floating point value being displayed has no fractional part, then the value is displayed without the decimal point or any decimal digits.

If the floating point number has six or less digits altogether, then all the digits are displayed. However, if the number of digits altogether is greater than six then those after the sixth digit are lost: the sixth digit is rounded (up or down) as appropriate.

(The total number of digits does not include the sign if it is a negative number.)

Discussion

Six is the magic number. It is called the default precision and is used to decide whether a number should be displayed in scientific notation or not (if the number of digits before the decimal point is greater than the default precision). It is also used to determine whether rounding should take place (if the total number of digits is greater than the default precision).

We can change the default precision by calling the `precision` member function of `cout`.

Subactivity 6.c.ii

Change the precision of the program in the previous subactivity to four. Hint: Insert the statement `cout.precision(4);` in the program.

Subactivity solution

```
//Illustrates setting the precision
#include <iostream>
using namespace std;

int main( )
{
   float x;

   cout << "Enter a floating point number: ";

   cin >> x;

   cout.precision(4);
   cout << "The number is " << x << endl;

   return 0;
}
```

Test this program with a few values to see the effect of the added statement.

Discussion

Note that the precision will now be set to 4 for all lines of code below this one. In other words, if we were to add a few more `cout` statements outputting floating point values below this, all of them would use a precision of 4.

Subactivity 6.c.iii

Now insert this statement just before the one setting the precision:

```
cout.setf(ios::fixed);
```

Then change the precision to a few different values, compiling and running the program each time to see what effect these changes have.

Subactivity solution

This statement prevents the computer from displaying floating point numbers in scientific notation. They are only displayed in fixed point notation.

Furthermore, it causes `cout` to use the current precision value to determine how many digits to display after the decimal point.

Discussion

There is a long explanation of the `setf` member function of `cout` which we won't go into here. Suffice it to say that `setf` stands for **set f**lags. There are a number of "flags" attached to `cout` that can be set or unset, and which affect how `cout` works. The flag we are interested in is `fixed`. We have to append `fixed` with `ios::` because it is defined in the `ios` class.

Note that the `fixed` flag remains in effect for insertions into the output stream following it.

You should now be able to write the program for the main activity.

Activity solution

```
//Calculates the sum of three prices and adds 14% VAT
#include <iostream>
using namespace std;

int main( )
{
   const float VAT_RATE = 0.14;
   float price1, price2, price3, total, vat, vatIncl;
```

```
    //Input three prices
    cout << "Enter the prices of three items" << endl;
    cin >> price1 >> price2 >> price3;

    //Calculate total, VAT and total including VAT
    total = price1 + price2 + price3;
    vat = total * VAT_RATE;
    vatIncl = total + vat;

    //Output the total including VAT
    cout.setf(ios::fixed);
    cout.precision(2);
    cout << "The total price is R" << vatIncl << endl;

    return 0;
}
```

Important points in this lesson

Programming concepts

In this lesson we have seen how to declare floating point variables, do calculations with floating point values, and store values in floating point variables.

We've seen what happens when we mix integers and floating point values. In particular, if we provide a floating point value for either (or both) operand(s) of an operator (i.e. `+`, `-`, `*` or `/`) then the result of the operation will be a floating point value.

We've also seen how to convert from integers to floating point values and vice versa. We can convert an integer to floating point by an *implicit type conversion*, i.e. by storing an integer value in a floating point variable. Attempting to assign a floating point value in an integer variable will cause the compiler to complain.

We can also convert from one type to the other (in either direction) by means of an *explicit type conversion*. We do this by putting the value we want to convert in parentheses and preceding it by the name of the type that we want to convert to. For example, `float(i)` will convert the value in integer variable `i` to a floating point value. Note, it won't change the value or the type of variable `i`, it will just return a floating point value.

We can format the output of floating point numbers in fixed point notation by using member functions of `cout` (also defined in the standard header file `iostream`). We looked at the following member functions: `precision` and `setf` with the `fixed` flag.

By default, C++ outputs floating point numbers in fixed point or scientific notation, depending on whether the number fits into the current precision. (The default precision is 6, but this can be changed using the `precision` member function.) If the `fixed` flag is set, only fixed point notation is used. Then the current precision is used to determine how many digits after the decimal point will be displayed.

Programming principles

- Test your programs thoroughly with a number of test cases. If necessary, use additional `cout` statements at intermediate points in the program to show the values of intermediate calculations.

- Rather use *explicit* than *implicit* type conversions. This makes it clearer what the program is actually doing.
- Use comments in your programs to explain what sections of code do. This is so that anyone who reads your program can easily follow what every part of the program does.

Exercises

Exercise 6.1

Write a program that asks for and reads the length and width of a room in metres. It then calculates and displays the area of the room with an appropriate heading. The output must be given in fixed-point notation, with three digits after the decimal point. Test your program with different inputs and make sure it gives the correct output.

Exercise 6.2

Change the program in Exercise 6.1 above so that it also calculates the total price of a wall-to-wall carpet for the room, at R59.50 per square metre.

Exercise 6.3

The following program does not do anything meaningful; it merely illustrates the use of integer and floating point numbers:

```
1  //Precision practice
2  #include <iostream>
3  using namespace std;
4
5  int main( )
6  {
7     int w, x;
8     float y, result, answer;
9
10    cout << "Enter two integers: " << endl;
11    cin >> w >> x;
12
13    y = w / x;
14    result = w - y;
15    y = float(w + x);
16    answer = int(y / x);
17
18    return 0;
19 }
```

(i) State where implicit and explicit type conversions take place.

(ii) What will the values of variables `result` and `answer` be at the end of the program if 13 and 5 are input for `w` and `x`?

Exercise 6.4

Consider the following program:

```
//Precision practice
#include <iostream>
using namespace std;

int main( )
{
   float a, b, c;

   cout << "Enter three floating point numbers:" << endl;
   cin >> a >> b >> c;

   cout.precision(5);
   cout << a << " " << b << " " << c << endl;

   cout << "Enter two more floating point numbers:" << endl;
   cin >> b >> a;

   cout.setf(ios::fixed);
   cout.precision(3);
   cout << a << " " << b << " " << c << endl;

   return 0;
}
```

What will the output of program be, given the following input? Show the precise formatting including spaces.

```
14.0      1.123      64.9999
73.46     27.2727
```

Lesson 7

String and character variables

Purpose of this lesson

In the lessons thus far, we have seen that we can declare integer and floating point variables to input and store integer and floating point values. But what about strings of characters? We know how to output messages, but what if we want to allow the user to input a string of characters? In this lesson we see how to declare string variables and input values into them.

We also see how to concatenate (i.e. join) two string values together.

Strings are made of individual characters. In the last two activities of this lesson we look at how to declare variables for storing individual characters and how to work with them.

Activity 7.a

Write a program to input a person's title, first name and surname and output the full name in the form *`Surname, Title, FirstName`*.

For example, say the user enters `Mr` for the title, `Koos` for the first name and `van der Merwe` for the surname. Then the program must output `van der Merwe, Mr, Koos`.

Test yourself

You will need to know the following to be able to write this program:

(i) How to declare string variables

(ii) How to input values into string variables

Concept (i) is simple enough. Concept (ii) turns out to be quite tricky, especially if you want to input strings that contain space characters. The subactivities below cover everything you need to know to be able to write the program.

Subactivity 7.a.i

It's easy to declare a string variable. Instead of using `int` or `float`, use `string`. The only complication is that you have to include the standard header file called <`string`> to be able to use the string type.

Inputting a string into a `string` variable is also easy. It works the same as for `int` and `float`.

With this information, write a program to perform the following interaction with the user:

```
Please enter your name: Rosemary
Hi Rosemary, pleased to meet you!
```

Subactivity solution

```
//Inputs and displays a string
#include <iostream>
#include <string>
using namespace std;

int main( )
{
   string name;
   cout << "Please enter your name: ";
   cin >> name;
   cout << "Hi " << name << ", pleased to meet you!" << endl;
   return 0;
}
```

Try running this program again, but enter `Mary Lou Fisher Price` for the name. The output may surprise you. The next subactivity is designed to show you how to solve this problem.

Subactivity 7.a.ii

Replace the statement `cin >> name;` with `getline(cin, name, '\n');` in the above program and run it again.

Discussion

The problem with the `>>` operator is that it inputs up to the first space character or newline character (i.e. the <Enter> key) when it is used to input a string. The space and newline characters are called *delimiters* because they are used to decide where the limit (i.e. the end) of a string is.

We use the `getline` function to specify an alternative delimiter, in this case '\n' (which represents the newline character alone). We can describe what the statement `getline(cin, name, '\n');` does as follows: It extracts all the characters up to the first occurrence of `'\n'` from `cin` and stores this string in the variable called `name`.

Subactivity 7.a.iii

Type in the following program, but before executing it, predict what you think will happen when you run it. Then run it and see if your prediction was correct.

```
//Illustrates the getline function
#include <iostream>
#include <string>
using namespace std;

int main( )
```

```
{
   int n;
   string s;
   cout << "Enter a number: ";
   cin >> n;
   cout << "Enter a sentence: ";
   getline(cin, s,'\n');

   cout << "The number is " << n << endl;
   cout << "The sentence is " << s << endl;
   return 0;
}
```

Subactivity solution

Woe, o woe! For some reason, `getline` doesn't seem to work after `>>`. The following subactivity should give you a clue as to why not.

Subactivity 7.a.iv

Type in the following program, but before executing it, predict what you think will happen when you run it. Then run it and see if your prediction was correct.

```
//cin.get( ); demo
#include <iostream>
using namespace std;

int main( )
{
   int n;
   cout << "Enter a number: ";
   cin >> n;
   cin.get( );
   cout << "The number is " << n << endl;
   return 0;
}
```

You might like to page back to Activity 4.b of Lesson 4 where we used `cin.get( );` previously.

Subactivity solution

You were thinking logically if you predicted that the computer would require you to press <Enter> again after entering the requested number. Surprisingly, however, the computer seems to ignore the statement `cin.get( );` and goes straight on to output the final message.

The reason is that `cin.get( );` extracts a single keystroke from the input stream, and that keystroke is the <Enter> that the user pressed after entering the integer for variable `n`. In other words, the statement `cin >> n;` reads the integer from `cin` (from the keyboard) but does not read the newline character (<Enter>).

Now, if you put two and two together you should be able to work out why `getline` doesn't seem to work after `>>`, and how you can fix the problem.

Subactivity 7.a.v

Fix the program in Subactivity 7.a.iii so that it uses `>>` to input the number and `getline` to input the sentence.

Subactivity solution

```
//Illustrates how to use >> and getline in a program
#include <iostream>
#include <string>
using namespace std;

int main( )
{
   int n;
   string s;
   cout << "Enter a number: ";
   cin >> n;
   cin.get( );
   cout << "Enter a sentence: ";
   getline(cin, s,'\n');
   cout << "The number is " << n << endl;
   cout << "The sentence is " << s << endl;
   return 0;
}
```

Discussion

This program illustrates a standard trick that you have to use if you want to use `getline` after `>>`. The statement `cin.get( );` removes the newline character left behind by `>>` so that `getline` can input up to the next newline character.

We have incorporated the concepts covered in the above subactivities in our solution to the main activity. Before you look at our solution, try to write it yourself. If you get stuck, go over the above subactivities again and try again. Then compare your solution with ours.

Activity solution

```
//Rearranges the title, first name and surname as input
#include <string>
#include <iostream>
using namespace std;

int main( )
{
   string title, firstName, surname;

   //Input the name
```

```
   cout << "Enter the title: ";
   cin >> title;
   cout << "Enter the first name: ";
   cin >> firstName;
   cin.get( );
   cout << "Enter the surname: ";
   getline(cin, surname, '\n');

   //Output the rearranged name
   cout << "The rearranged name is ";
   cout << surname << ", " << title << ", " << firstName << endl;
   return 0;
}
```

Discussion

The final `cout` statement in the above program is somewhat unsatisfactory since it does all the processing required by the problem (namely to rearrange the parts of the name). This is similar to the situation where we were doing calculations in a `cout` statement. In that situation, we said that it was better to introduce a separate variable, store the result of the calculation in the variable (using an assignment statement) and then output the value of the variable.

We can do something similar here. We can introduce an additional string variable and store the rearranged name in it. The only problem is that we can't use the `<<` operator to join the parts of the name together. To concatenate (i.e. join) two strings together, we use the `+` operator. This sometimes causes confusion because it looks just like addition when applied to integer or floating point values, but C++ chooses the appropriate `+` operator depending on the operands (i.e. the values on either side) that it is given.

To apply this idea to the program above, we would have to declare an additional string variable (say `fullName`). Then we would concatenate the parts of the name together and store the result in it as follows:

```
fullName = surname + ", " + title + ", " + firstName;
```

Finally we could output the rearranged name with the single statement

```
cout << "The rearranged name is " << fullName << endl;
```

Activity 7.b

The following program inputs a single letter and outputs the next letter in alphabetic order. Study it and answer the question that follows:

```
//Next letter in alphabetic order
#include <iostream>
using namespace std;
```

```
int main( )
{
   char letter, next;

   cout << "Enter a letter of the alphabet: ";
   cin >> letter;

   next = letter + 1;
   cout << "The next letter is " << next << endl;

   return 0;
}
```

What is going on in the statement `next = letter + 1;`?

Test yourself

Before you try to answer the question, note the declaration of variables `letter` and `next`. The `char` type is used to store individual characters, as opposed to the `string` type, which is used for multiple characters.

Although the `string` type can be used to store a single character, you can't do what the above program does with a string. (To see this, change the declaration of `letter` and `next` to `string` and try compiling the program.)

The answer to the question is not straightforward. Implicit type conversions are taking place, and you have to identify and explain them.

The two subactivities below illustrate the funny things that are going on.

Subactivity 7.b.i

Type in the program of Activity 7.b but remove variable `next`. You will also have to change the output statement to

```
cout << "The next letter is " << letter + 1 << endl;
```

Predict the output.

Subactivity solution

Weird! The program outputs an integer.

Discussion

What integer is it that is output? Every character (not just the letters of the alphabet - all characters including punctuation marks and the numerical digits '0' to '9') have an integer associated with them.

This integer is called its ASCII code. Appendix C (at the end of this guide) contains a table of the ASCII codes of all the possible characters. `letter + 1` therefore represents 1 more than the ASCII code of `letter` (whatever character it represents).

But there is something else funny going on here. The `+` operator is being used between two values of different types, a `char` and an `int`. Remember that we have seen various versions of the `+` operator: one for integers (Lesson 2), one for floating point numbers (Lesson 6), and one for strings (end of Activity 7.a of this lesson). Is there perhaps another version of `+`, namely for chars? The answer is No. The output is an integer, showing us that the integer version of `+` is being used. We conclude that an implicit type conversion of the first operand is taking place, namely from a `char` to an `int`. The character is converted implicitly to its ASCII code, which is an integer.

This program gives another reason why it is a good idea to introduce a variable (eg. `next`) rather than perform the calculation in the `cout` statement.

Subactivity 7.b.ii

Explain how you would change the program below to get rid of variable `c`. The program should give the same output.

```
//ASCII converter
#include <iostream>
using namespace std;

int main( )
{
   char c;
   int i;
   cout << "Enter an integer: ";
   cin >> i;
   c = i;
   cout << i << " represents the character " << c << endl;
   return 0;
}
```

Hint: Use an explicit type conversion.

Subactivity solution

Remove the declaration of variable `c`, remove the assignment statement and replace `c` in the output statement with `char(i)`.

Discussion

The assignment statement `c = i;` in the program above uses an implicit type conversion. Variable `i` is of type `int`, and variable `c` is of type `char`. The integer value of `i` is converted implicitly (behind the scenes) to a character to be stored in variable `c`.

We can perform explicit type conversions from `int` to `char` (or vice versa) in the same way as we do explicit type conversion between integers and floating point numbers. We simply use the type name `char`, followed by an integer value in parentheses.

Now attempt to complete the main activity. You need to identify all the implicit type conversions taking place.

Activity solution

There are two implicit type conversions taking place in the statement `next = letter + 1;`

Firstly, the value of `letter` (a `char`) is implicitly converted to an integer to be able to calculate `letter + 1` (because integer addition is used). Since `letter + 1` is an integer, another implicit type conversion, this time from `int` to `char`, is performed when the result of the calculation is assigned to variable `next`.

Activity 7.c

Change the program developed in Activity 7.a to display the initial of the first name followed by a fullstop, in place of the full first name.

For example, if the person enters `Mr` for the title, `Koos` for the first name, and `van der Merwe` for the surname, the program should output

```
van der Merwe, Mr, K.
```

Test yourself

There are many ways of extracting a single character from a string, some of which are explained in the lesson entitled **String manipulation** near the end of this guide. For now, we are going to use a little trick which stems from the fact that a string consists of a number of characters. By inputting into a `char` variable when the user enters a string, we can separate the first letter from the rest of the string. The subactivity below shows how this can be done.

Subactivity 7.c.i

Before typing in the following program, predict what its output will be:

```
//Rotates an integer
#include <iostream>
using namespace std;

int main( )
{
   char c;
```

```
   int i;
   cout << "Enter an integer greater than 9: ";
   cin >> c >> i;
   cout << "Rotated it is " << i << c << endl;
   return 0;
}
```

Subactivity solution

The output consists of the first digit of the number removed and pasted onto the rest of the digits.

The user is fooled into thinking that the program is only inputting a single value, when in fact it inputs two values, a character, and an integer.

Discussion

Remember our discussion previously about delimiters? These are characters that indicate the end of a value. In general, the delimiters for integers, floating point numbers and strings are the space character and the newline character.

Here we see that there is no delimiter for inputting character values. If we input into a `char` variable, only one character is input, and the rest of the input is left in the input stream for the next value to be input (into whatever type of variable).

Apply the above idea to solve the problem of the main activity.

Activity solution

We used the revised solution as explained in the **Discussion** section after the solution to Activity 7.a:

```
//Rearranges the title, first name and surname as input
#include <string>
#include <iostream>
using namespace std;

int main( )
{
   string title, firstName, surname, fullName;
   char initial;

   //Input the name
   cout << "Enter the title: ";
   cin >> title;

   cout << "Enter the first name: ";
   cin >> initial >> firstName; //trick!
   cin.get( );
```

```
    cout << "Enter the surname: ";
    getline(cin, surname, '\n');

    //Output the rearranged name
    fullName = surname + ", " + title + ", " + initial + ".";
    cout << "The rearranged name is " << fullName << endl;
    return 0;
}
```

Discussion

Here are a few comments on this program:

We saw previously that `cin.get( );` inputs a single character, but that it throws away the character that it gets. We now know that there is a way of inputting a single character and keeping it, namely by using `cin` with a `char` variable. There is (another) important difference between `cin.get( );` and inputting into a `char` variable, however. We can't use `cin` with a `char` variable to get rid of a newline character sitting in the input stream, as `cin.get( );` does. You might like to try replacing `cin.get( );` with a statement that inputs a value into a `char` variable. (You'll have to declare another variable, say `char c;`). Some strange things happen.

Note secondly that we are concatenating a `char` to a `string` with the expression `", " + initial`. Once again this is an example of an implicit type conversion, this time from a `char` to a `string`, because we can see that the string version of the `+` operator (concatenating two strings) is being used.

We used `"."` to specify the fullstop to be appended to the initial. In general, we prefer to use single quotes when we want to specify a single character, and double quotes for more than one character (i.e. a string). In this case, however, we have to use double quotes (even though it is a single character) because we are concatenating strings. When used to concatenate strings, the `+` operator requires two strings as operands, and the compiler can get confused if one of the operands is a `char` value.

Just to confuse you, note the use of single quotes in the previous line, namely `getline(cin, surname, '\n');` Here there are two characters between single quotes! In fact `'\n'` represents a single character, namely the newline character. We need a special way of referring to the newline character since there isn't a normal character that represents it. This is an example of an *escape character*, and we will see an example of another escape character in the following activity.

Activity 7.d

Write a program to input a number of kilometres and a number of litres (both floating point numbers) and to calculate the fuel consumption in two ways: kilometres per litre and litres per 100 kilometres.

The output must be displayed in the following way:

```
kms     litres  km/l    l/100km
871     55.4    15.722  6.36051
```

Test yourself

The calculations are easy enough. We do not give any subactivities to help you with them. The subactivities provided below show how to display headings and values in tabular columns. Before working through them, we suggest that you write a program to perform the necessary calculations and display the answers on separate lines.

Then after doing the subactivities, reload your program and apply the new ideas.

Subactivity 7.d.i

Explain the change caused in the output of the following program when `' '` is replaced by `'\t'` in the second `cout` statement.

```
//Displaying in columns
#include <string>
#include <iostream>
using namespace std;

int main( )
{
   int x, y;
   cout << "Enter two numbers: ";
   cin >> x >> y;
   cout << x << ' ' << y << endl;
   return 0;
}
```

Subactivity solution

If we replace the second `cout` statement with

```
   cout << x << '\t' << y << endl;
```

a number of spaces are displayed between the two integers.

Discussion

The escape character `\t` causes the cursor to jump to the next tabular column if it is inserted in the output stream. Tabular columns are generally spaced 8 characters apart.

Subactivity 7.d.ii

In your study notebook, write a `cout` statement to output the three strings `Red`, `Green` and `Blue` in tabular columns. The `cout` statement may not use any variables.

Subactivity solution

There are several solutions to this question. We give two different ones.

One solution alternatively inserts strings and the escape character '\t' into cout:

```
cout << "Red" << '\t' << "Green" << '\t' << "Blue" << endl;
```

Another solution includes the escape character \t in a single string which is inserted into cout:

```
cout << "Red\tGreen\tBlue" << endl;
```

As we have stated earlier, a string is simply a collection of characters. From the second solution you can see that a string can contain escape characters.

Discussion

Note that an escape character is a single character value, even though it is specified by two character symbols. A escape character always starts with \, and is followed by another character that must be interpreted in a different way as it would be normally.

Here are two other useful escape characters:

- \" This represents the double quote character. This is necessary if you want to include the double quote character inside a string. It must be an escape character because without a preceding \, a " will be interpreted as the beginning or end of a string.
- \\ This represents the backslash character \. You always have to use two backslashes if you want one, because a single backslash will be interpreted as signifying an escape character to follow.

Appendix B specifies a few more escape characters.

You should now be able to complete the main activity. If you haven't written a program to do the calculations yet, you will have to write the whole program from scratch, including displaying the values in tabular columns.

Activity solution

```
//Calculates fuel consumption, displayed in tabular columns
#include <iostream>
using namespace std;

int main( )
{
   float kilometres, litres, kmpl, lp100km;

   //Input kilometres and litres
```

```
    cout << "Enter the Kilometres: ";
    cin >> kilometres;
    cout << "Enter the litres: ";
    cin >> litres;

    //Calculate consumptions
    kmpl = kilometres / litres;
    lp100km = litres / (kilometres / 100);

    //Output consumptions in columns
    cout << "kms\tlitres\tkm/l\tl/100km" << endl;
    cout << kilometres << '\t' << litres << '\t';
    cout << kmpl << '\t' << lp100km << endl;

    return 0;
}
```

There are alternatives for the formula for calculating the litres per 100 kilometres, namely

```
    lp100Km = litres / kilometres * 100;
```

or

```
    lp100km = 100 / kmpl;
```

Important points in this lesson

Programming concepts

We can declare string variables like any other variables. We just have to include the standard header file `<string>`. (This is because the `string` type is not a primitive type like `int`s, `float`s and `char`s.)

The only operator that is available for string values is the `+` operator. It is used to concatenate two strings together. The compiler decides which version of `+` to use (for `int`s, `float`s or `string`s) by the operands provided on either side of it. (By the way, if you provide an integer as the one operand of `+` and a string as the other, the compiler will complain. No attempt will be made to implicitly convert one type to the other.)

We can use a `cin` statement to input a value into a string variable. However, we have to use `getline` to input a string containing spaces into a single string variable. The format of a `getline` statement is

```
getline(cin, StringVariable, Delimiter);
```

where *`StringVariable`* is declared as string, and *`Delimiter`* is a single character, usually the newline character `'\n'`, to specify that everything up to that character must be stored in *`StringVariable`*.

We have seen that the `>>` operator does not remove the newline character from the input stream when used with `cin`. (It gets everything up-to-but-not-including the newline character.) This causes a problem when `getline` is used after `>>` - `getline` doesn't input anything because it immediately encounters a newline character. To work around this problem, we use `cin.get( );` to remove the newline character left in the input stream after `>>`, before we use `getline`.

A string is made up of characters. There is a primitive type called `char` that we can use to declare a variable to store a single character. We can input a single character into such a variable, and output single characters. We use single quotes to specify a character literal, and double quotes to specify a string literal. If a string literal consists of a single character, we prefer to specify it as a `char` with single quotes.

If a character value is used as an operand of the `+` operator, it is implicitly converted to the type of the other operand. In other words, if a character is used with an integer, the character is implicitly converted to an integer (its ASCII code) and integer addition is performed. If it is used with a string, it is implicitly converted to a string and string concatenation is performed.

If an integer value is assigned to a character variable, the integer value is implicitly converted to a character (according to the ASCII code that it represents).

The \ character is used to indicate an escape character. The character following it is given a special meaning, i.e. other than its normal meaning. The escape characters that we have learnt about are:

- `\t` This represents the *tab character*. It will cause the cursor to jump to the next tab column if it is inserted in the output stream.
- `\n` This represents the *newline character*. If it is inserted into the output stream, it will cause the cursor to jump to the next line.
- `\"` This represents the *double quote character*. It is necessary if you want to include the double quote character inside a string. It is a escape character because without a preceding backslash, a `"` will be interpreted as the beginning or end of a string.
- `\\` This represents the *backslash character*. You always have to use two backslashes if you want one, because a single backslash will be interpreted as signifying an escape character to follow.

Programming principles

When writing programs, you will be faced with the decision of what variables to declare and what their types should be. If you only need to store whole numbers, you should declare `int` variables. If you need to calculate and store floating point values, declare `float` variables. If you need to store strings of characters, use `string` variables, but if you only need to store single characters, use `char` variables.

Sometimes, your first choice of variables and their types is insufficient for the purposes of the program. You should be prepared to change your plans, i.e. change the types of variables or declare additional ones.

Exercises

Exercise 7.1

Consider the following program:

```
//The + operator
#include <iostream>
using namespace std;

int main( )
{
   int x, y, z;
   cout << "Enter two numbers for variables x and y: ";
```

```
    cin >> x >> y;

    z = x + y;
    cout << "x + y is " << z << endl;

    return 0;
}
```

(i) What will the output of the program be if the user enters 123 and 456 for the two variables?

(ii) If we change the declaration of variables `x`, `y` and `z` to `string` (we'll also need to add `#include <string>` to the beginning of the program), what will the output of the program be if the user enters 123 and 456 for the two variables?

(iii) What will the output be if we change the declaration of `x`, `y` and `z` to char?

Exercise 7.2

Write a program that works out which letter of the alphabet a given upper case letter represents. The user interface should look as follows:

```
Enter an upper case letter: E
E is in position 5 in the alphabet
```

Exercise 7.3

Write a program to perform a "spoonerism". A spoonerism is where the first letters of two words are swapped around. For example, the spoonerism of `Cold Blue` is `Bold Clue`. The program must input two words and output their spoonerism.

Exercise 7.4

Write a program to allow the following interaction with the computer:

```
Enter a person's name: Peter
Enter another person's name: Mary
Enter a colour: purple
Enter a number: 13
Enter a noun: fish
Enter an adjective: sweet

Dialogue
========
Peter:  "Couldn't you see that the traffic light was purple?"
Mary:   "But I had 13 people and a fish in the car with me."
Peter:  "That is so sweet! You could have had them all killed."
```

Exercise 7.5

Write a program to perform the following interaction with the user:

```
Computer punishment
-------------------
```

```
Repetitions? 10000
Message? I will not drive humans crazy again!

I will not drive humans crazy again!
I will not drive humans crazy again!
I will not drive humans crazy again!
:
I will not drive humans crazy again!
```

The message should be displayed as many times as requested by the user; in this case, 10000.

Part II

Conditional execution

There are two aspects of computer programs that make them very powerful. Firstly, a good program can deal with general tasks, or at least a whole range of similar tasks. Secondly it can do repetitive tasks very quickly.

If you think about it, the programs we have written so far have not been able to do either of these things very well. Each program could only do a single task, or perhaps very slight variations on a task (e.g. a calculation dependant on the user's input). Also, if we wanted the program to perform the same sequence of instructions over and over, we would have to repeat the instructions, which would make a program very long.

To get a program to perform a larger range of tasks, we need to get it to be able to make decisions. This might sound like getting a computer to think like a human, but it's much simpler than that. For example, say the purpose of a program is to report whether a test mark received as input is a 'pass' or a 'fail', it will display either a message saying it is a pass, or a message saying it is a fail. The `cout` statements for both messages will appear in the program, but depending on the input, only one of them should be executed. *Conditional statements* allow a computer to make these kinds of decisions and do different things under different conditions.

Iteration statements allow us to specify a sequence of statements that should be repeated, either a fixed number of times, or while certain conditions hold. In this way, we can get a computer program to execute many instructions without us having to write too many lines of code.

Lesson 8

If statements

Purpose of this lesson

Conditional statements are used to allow a computer program to do different things under different conditions. The first type of conditional statements that we are going to consider are `if` statements. They are used to perform statements dependant on a single condition. A condition is often expressed as a comparison between a variable and a value, e.g. `x > 10`, or between two variables or between two complicated expressions. In this lesson we look at `if` statements and conditions in C++.

Activity 8.a

Write a program that asks for and inputs an employee's yearly salary, and then calculates and displays the income tax payable on the salary. If the salary exceeds R 70 000.00 per year, the tax rate is 40%, otherwise it is 30%.

Test yourself

If you managed to write the program successfully, you may skip the subactivities that follow and compare your solution with ours. Did you use an `if` statement? If not, or if your program differs substantially from ours, read through the subactivities below to see how we approached the problem.

Subactivity 8.a.i

Write a program that asks for and inputs an employee's yearly salary. It then calculates and displays 40% income tax on that amount. (No `if` statement is required for this program.)

Subactivity solution

To calculate 40% of the salary, we multiply it by 0.40. Since the tax rate is given in the problem definition, we can include it as a constant in our program.

The program looks as follows:

```
//Calculates tax on a salary
#include <iostream>
```

```
using namespace std;

int main( )
{
   const float TAX_RATE = 0.40;
   float salary, tax;

   //Ask for and input the salary
   cout << "Enter the employee's salary: ";
   cin >> salary;

   cout << endl;  //To display a blank line

   //Calculate and display the income tax
   tax = salary * TAX_RATE;
   cout << "The tax on a salary of R" << salary;
   cout << " is R" << tax << endl;

   return 0;
}
```

Subactivity 8.a.ii

This subactivity covers what is still needed to complete the solution for the main activity.

Let us look again at what the program should do. As before, it asks for the salary and inputs it. Then it should check whether the salary is greater than 70 000.00. If so, the income tax must be calculated as 40% of the salary, otherwise it must be calculated as 30% of the salary.

For this, we need an `if` statement. Below we give an incomplete `if` statement. Using your study notebook, fill in the missing statements to calculate the appropriate tax:

```
if (salary > 70000.00)
   - - - - - - - - - - - - - - - - -
else
   - - - - - - - - - - - - - - - - -
```

Subactivity solution

There are a number of correct solutions to this subproblem. One solution would be

```
if (salary > 70000.00)
   tax = salary * 0.40;
else
   tax = salary * 0.30;
```

Or, if we use constants for the respective tax rates:

```
if (salary > 70000.00)
   tax = salary * HIGH_TAX;
else
   tax = salary * LOW_TAX;
```

If we are not using constants, another possible solution is:

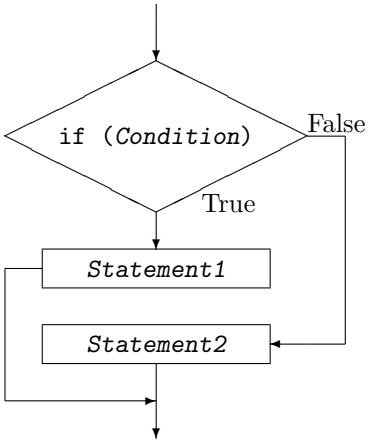
```
if (salary > 70000.00)
   taxRate = 0.40;
else
   taxRate = 0.30;
tax = salary * taxRate;
```

Discussion

The general structure of an `if` statement is:

```
if (Condition)
   Statement1;
else
   Statement2;
```

We can represent this diagrammatically as in the flow chart alongside:



If the *Condition* is True, *Statement1* is executed but if the *Condition* is False, *Statement2* is executed.

The *Condition* is something that can either be True or False. It usually involves a *relational operator*, i.e. one of the following:

`<`	is less than
`<=`	is less than or equal to
`>`	is greater than
`>=`	is greater than or equal to
`==`	is equal to
`!=`	is not equal to

Examples of conditions are:

`price > 20.00`	True if `price` is greater than 20.00
`value != 0`	True if `value` is not equal to 0
`x >= y`	True if `x` is greater than or equal to `y`

In a later lesson we will look at more complicated forms that *Condition* can take. For now, you only need to consider simple conditions such as those given above.

Subactivity 8.a.iii

You can now modify the program of Subactivity 8.a.ii so that it solves the problem of the main activity.

Activity solution

```
//Calculates tax on a salary
#include <iostream>
using namespace std;

int main( )
{
   const float HIGH_TAX = 0.40;
   const float LOW_TAX = 0.30;
   float salary, tax;

   //Ask for and input the salary
   cout << "Enter the employee's salary: ";
   cin >> salary;

   //Calculate and display the income tax
   if (salary > 70000.00)
      tax = salary * HIGH_TAX;
   else
      tax = salary * LOW_TAX;

   cout << endl << "The income tax on a salary of ";
   cout << salary << " is " << tax << endl;

   return 0;
}
```

Discussion

The layout an `if` statement is important. Note that in our program, the `if` and `else` reserved words are aligned, and the statements under them are indented. This ensures that anyone who reads the program will immediately see what falls under the which part.

To illustrate how important this is, here are some examples of `if` statements that the computer will accept and execute correctly, but that are more difficult to read:

```
   if (salary > 70000.00) tax = salary * HIGH_TAX; else
   tax = salary * LOW_TAX;
```

and

```
   if (salary > 70000.00)
```

```
tax = salary * HIGH_TAX;
else tax = salary * LOW_TAX;
```

Activity 8.b

The SupaDupa clothing store has a sale on. For any items less than R200.00, the standard discount is 10%. For items worth R200.00 or more, the discount must be decided individually by the store manager.

Write a program that inputs the price of an item, calculates the discount, and then displays the old and new price in the form of a tag which can be printed and attached to the item. Here is an example of the user interaction:

```
Enter old price: 289.00
Enter special discount %: 15

=============
Was: R289.00
Discount: 15%
Now: R245.65
=============
```

Test yourself

This program also requires an `if` statement to decide whether to prompt the user for a discount or not. The difference is that the program needs to do more than one thing (i.e. execute more than one statement) if the condition of the `if` statement is True. The trick is to put all these statements in braces (curly brackets). If you need help in writing it, work through the subactivities below, otherwise go on to where we give the solution.

Subactivity 8.b.i

Study the problem definition of the main activity and make a list in your study notebook of the things that you think need to be done in the program.

Subactivity solution

We came up with the following steps that the program must perform:

- *Ask for and input the price of the item.*
- *Decide whether a special discount must be input, and if so, input it. Otherwise use the standard discount.*
- *Calculate the discount amount as a percentage of the price.*
- *Calculate the final price by subtracting the discount amount from the old price.*
- *Display the old price, the discount percentage and the final price.*

How you divide the problem into steps depends very much on you, the programmer. Your list may look different to ours. For example, you might have combined calculating and subtracting the discounted amount to get the final price into one step. As we discuss below, there are many correct ways of solving this problem.

Subactivity 8.b.ii

The second step is probably the most complicated. In your study notebook, write down an `if` statement that uses a variable called `price` and a constant called `CRITICAL_VAL` to decide whether or not to input a special discount. If `price` is greater than or equal to `CRITICAL_VAL`, input a special discount into a variable called `discount`. Otherwise, store the value of another constant called `STANDARD_DISCOUNT` in variable `discount`.

Subactivity solution

Our `if` statement looks like this

```
if (price >= CRITICAL_VAL)
{
   cout << "Enter the special discount: ";
   cin >> discount;
}
else
   discount = STANDARD_DISCOUNT;
```

Note how we used braces to get more than one statement to be executed if the condition is True. We don't use braces for the `else` part because there only one statement that needs to be executed for it.

Subactivity 8.b.iii

The rest of the program is fairly straightforward. Write the entire program now. Don't forget to declare all the necessary constants and variables.

Activity solution

```
//Calculates discount on a price and displays a tag
#include <iostream>
using namespace std;

int main( )
{
   const float CRITICAL_VAL = 200.00;
   const int STANDARD_DISCOUNT = 10;

   int discount;
   float price, discountAmount, finalPrice;

   //Ask for and input the item's price
   cout << "Enter the price of the item: ";
   cin >> price;
```

```
   //Determine discount
   if (price >= CRITICAL_VAL)
   {
      cout << "Enter the special discount: ";
      cin >> discount;
   }
   else
      discount = STANDARD_DISCOUNT;

   //Calculate the final price by subtracting the discount
   discountAmount = price * discount / 100;
   finalPrice = price - discountAmount;

   //Display the tag
   cout << "===========" << endl;
   cout << "Was: R" << price << endl;
   cout << "Discount: " << discount << "%" << endl;
   cout << "Now: R" << finalPrice << endl;
   cout << "===========" << endl;

   return 0;
}
```

Discussion

In the previous activity of this lesson we saw that *`Statement1`* and *`Statement2`* of an `if` structure can be single C++ statements. In this activity, we have seen that they can also be *compound statements*, i.e. more than one statement enclosed between braces. This is quite useful when we need to execute more than one statement depending on the condition of the `if` statement.

Which statements to put where is not always so easy to decide, however. For example, we could have written the above `if` statement as follows:

```
   //Calculate discount
   if (price > CRITICAL_VAL)
   {
      cout << "Enter special discount: ";
      cin >> discount;
      discountAmount = price * discount / 100;
      finalPrice = price - discountAmount;
   }
   else
   {
      discount = STANDARD_DISCOUNT;
      discountAmount = price * discount / 100;
      finalPrice = price - discountAmount;
   }
```

Here we have duplicated the assignment statements that were after the `if` statement and put them inside the two parts of the `if` statement. Although this will work perfectly well, we hope you'll agree

that the original solution is much better. In fact, whenever you see that the same code is repeated in the two parts of an `if` statement, you should take the common code out and place it either before or after the `if` statement.

Also, not all `if` statements necessarily need an `else` part. Look at the following alternative to our solution:

```
//Calculate discount
discount = STANDARD_DISCOUNT;
if (price >= CRITICAL_VAL)
{
   cout << "Enter the special discount: ";
   cin >> discount;
}
```

Here we initialise `discount` to `STANDARD_DISCOUNT` and only change its value if necessary. In other words, if the price is less than the critical value, nothing further needs to be done, so the `else` part is omitted.

Important points in this lesson

Programming concepts

In this lesson we saw how to make decisions in a program using an `if` statement. The general form of an `if` statement is:

```
if (Condition)
   Statement1;
else
   Statement2;
```

The *Condition* can either be True or False. If *Condition* is True, *Statement1* is executed, but if *Condition* is False, *Statement2* is executed.

In the simple forms of `if` statements we have encountered thus far, *Condition* involves one of the following relational operators, applied to two floating point values:

`<` is less than

`<=` is less than or equal to

`>` is greater than

`>=` is greater than or equal to

`==` is equal to

`!=` is not equal to

These relational operators also work with integer and string values and variables.

Statement1 and *Statement2* can be single C++ commands, or compound statements. (A compound statement is where we combine a sequence of statements between braces.) For example,

```
if (Condition)
{
   StatementSequence1;
}
else
{
   StatementSequence2;
}
```

The `else` part can also be left out:

```
if (Condition)
   Statement;
```

or

```
if (Condition)
{
   StatementSequence;
}
```

Programming principles

Beware of the difference between the assignment operator `=` and the relational operator `==`. Many C++ programmers have shed many tears over the fact that they used `=` when they meant to use `==`. Their programs gave the wrong answers and they couldn't work out why.

To improve readability, we always put the `else` in line with (i.e. directly underneath) the `if`, and indent the statements within them to the right.

Exercises

Exercise 8.1

In which of the following pieces of code is *Statement1* executed?

(i)
```
if (absoluteZero == absoluteZero)
   Statement1;
else
   Statement2;
```

(ii)
```
if (absoluteZero < absoluteZero)
   Statement1;
else
   Statement2;
```

(iii)
```
if (absoluteZero != absoluteZero)
   Statement1;
Statement2;
```

(iv)
```
if (-1.0 < 0.0) Statement1; Statement2;
```

Exercise 8.2

Simplify the following `if` statements by taking out common code:

(i)
```
if (colour == "red")
{
   cout << "Correct" << endl;
   cout << "What is the colour of the sky? ";
   cin >> colour;
}
else
{
   cout << "No, blood is red." << endl;
   cout << "What is the colour of the sky? ";
   cin >> colour;
}
```

(ii)
```
if (age < 13)
{
   cout << "Enter salary: ";
   cin >> parentSalary;
   pocketMoney += parentSalary / 20;
}
else
{
   cout << "Enter salary: ";
   cin >> parentSalary;
   pocketMoney += parentSalary / 10;
}
```

(iii)
```
if (mark < 50)
{
   failed++;
   total += mark;
}
else
   total += mark;
```

Exercise 8.3

Write an `if` statement that enacts the following logic: 'If balance is greater than or equal to zero, display the word "Credit" on the screen; otherwise display the word "Debit"'.

Exercise 8.4

Design an `if` statement that tells the user whether or not a floating point variable `x` contains a value equal to that in another floating point variable `y`.

Exercise 8.5

Consider the following (horrible) piece of code:

```
int jd; const int BANANA_FISH = 1945; const float SILLY = 20.0;
cout << "When were you born? "; cin >> jd;
```

```
if (jd < BANANA_FISH) cout << "Free entry";
else cout << "Entrance fee R" << SILLY; cout << endl;
```

Rewrite the code using meaningful variable names, a comment statement and indentation to improve its readability.

Exercise 8.6

Write a program that inputs the length and width of a room. It then checks whether 100 square metres of carpet will be enough to cover the floor of the room. If it is enough, display a message that reports this; otherwise display the size of the area that will not be covered.

Lesson 9

While loops

Purpose of this lesson

In the previous lesson, we looked at the `if` statement. In this lesson, we introduce a way of controlling repeated execution, the `while` loop. The `if` statement allows a program to choose between two possible statements (or groups of statements). By contrast, the `while` loop lets a program execute the same statement (or group of statements) repeatedly, as long as a given condition is True.

Activity 9.a

We wrote the following program in a previous lesson. It inputs the prices of three items, adds them together, and finally adds the VAT to get the total price:

```
//Calculates VAT on three prices
#include <iostream>
using namespace std;

int main( )
{
   const float VAT_RATE = 0.14;
   float price1, price2, price3, total, vat, vatIncl;

   cout << "Enter the prices of three items" << endl;
   cin >> price1 >> price2 >> price3;
   total = price1 + price2 + price3;
   vat = total * VAT_RATE;

   vatIncl = total + vat;
   cout << endl;
   cout << "The total price is " << vatIncl << endl;

   return 0;
}
```

Suppose the user of the program is given a trolley full of items of which the total price must be calculated. This program will only work if there are exactly three items. How will a program look that can compute the sum of an unknown number of values?

Your task in this activity is to write a program that inputs the prices of zero or more items, and displays the total price with 14% VAT added. (Hint: The program will know when the last price has been input if a price of 0.00 is entered.)

Test yourself

If you completed the program successfully, compare it with our solution. If it differs and you are not sure whether yours is correct, or if you do not know how to tackle the problem, work through the subactivities below. Check that your program works correctly for the following test cases:

(i) Three items, costing 12.04, 2.99, and 0.96, followed by 0.00

(ii) Four items: 4.50, 12.20, 2.22, 18.75, followed by 0.00

(iii) No items (i.e. simply enter 0.00)

As you can check with a calculator, the answers to these three test cases should be 18.23, 42.94 and 0.00 respectively.

Subactivity 9.a.i

Think carefully about what the main activity's program should do. In your study notebook, write down (in English) the steps that you think the program must follow in order to solve the problem.

Subactivity solution

Here is our answer:

```
  Step 1. Display a message that prompts the user to enter the first value.
  Step 2. Input the first value.
  Step 3. Set the total price equal to the first price entered.
  Step 4. If the first price entered is not 0.00, input the price of the second item and
          add that to the total price so far; otherwise go to the last part of the program
          (step x).
  Step 5. If the second price entered is not 0.00, input the price of the third item and
          add that to the total price so far; otherwise go to step x.
        :
  Step 9. If the sixth price entered is not 0.00, input the price of the seventh item and
          add that to the total price so far; otherwise go to step x.
        :
  etc ...
        :
  Step x. Add 14% VAT to the total price.
Step x+1. Display the result.
```

Discussion

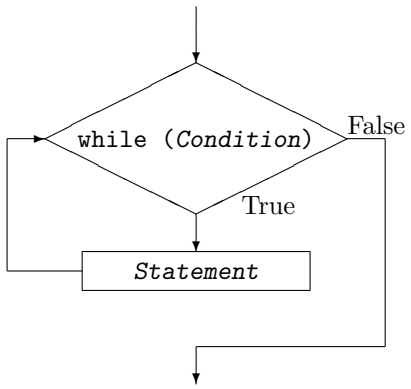
It is clear from this algorithm that the program will perform the same task repeatedly. Seeing that we have learnt how to use the `if` statement, one solution that springs to mind is to use a succession of `if` statements. But how many? We do not know beforehand how many items there will be, and with every execution of the program, the number of items might be different.

To solve a problem like this, we need an iterative (or looping) structure. The `while` loop is like a decision structure in that it allows a program to repeat execution of a group of statements as long as some condition is True. As soon as the condition becomes False, the repetition stops.

C++'s `while` loop has the following structure:

```
while (Condition)
   Statement;
```

We can represent this diagrammatically as in the flow chart alongside:



Statement is executed repeatedly while *Condition* is True. The only exit from the `while` loop occurs when *Condition* is False.

Condition has the same form as the condition of an `if` statement, and, as before, *Statement* can be a single statement or a compound statement. (We sometimes refer to *Statement* as the *body of the loop*.)

For the sake of readability we always indent *Statement*; so that it is clear which statement, or block of statements, will be repeated by the `while` loop.

Subactivity 9.a.ii

Let us see if we can put these principles into practice. What does the following code fragment do?

```
while (price != 0.0)
{
   cout << "Enter a price: ";
   cin >> price;
}
```

Subactivity solution

When this `while` loop is encountered for the first time, the program checks whether variable `price` has the value 0.0. If `price` is equal to 0.0, it skipa the whole body of the `while` loop and continues with the statement that follows the closing brace }. If `price` is not equal to 0.0, the body of the `while` loop is executed. This means a new value will be input for `price`. Execution then returns to the top of the `while` statement, and the condition `price != 0.0` is checked again. If `price` is still not equal to 0.0, the body is executed again. This process goes on until the user enters the value 0.0, in which case the condition becomes False.

Subactivity 9.a.iii

Let us try another example. What does this code do?

```
while (counter < 10)
   counter++;
```

Subactivity solution

This `while` statement checks whether `counter < 10`. If it is, `counter` is incremented by 1, and the program loops until `counter` has the value of 10. On the other hand, if `counter` is greater than or equal to 10, the loop body does not execute. In this case, the value of `counter` does not change and the program skips forward to the code that comes after the loop body.

Discussion

There are three very important aspects of the `while` statement that you should keep in mind when using it in a program. They have to do with the variable that is used in the condition of the `while` loop to control repetition. This variable is called the *control variable*.

1. Initialising the control variable

The variable (or variables) that appear in the condition must be initialised before entering the while loop for the first time.

As stated in previous lessons, an uninitialised variable can have any value. There are a few ways of ensuring that a variable has a value: (i) initialise it when it is declared, (ii) give it a value with an assignment statement, or (iii) input a value for it with `cin`.

In the first example above, the `while` loop starts with

```
while (price != 0.0)
```

Suppose we have a list of ten items for which we want to input prices using this `while` loop. If we do not input a value for `price` before reaching the loop for the first time, its value could be anything. This will probably mean that the condition is True and the body of the loop will be executed with some strange value in variable `price`.

To get the desired result, the program must input the first item's price before entering the `while` loop. We should therefore have something like this:

```
cout << "Enter the price of the first item: ";
cin >> price;
while (price != 0.0)
{
   cout << "Enter a price: ";
   cin >> price;
}
```

2. Testing the value of the control variable

The condition of the `while` loop must test whether the control variable has some or other value. The loop is repeated while the condition is True. The condition should therefore specify the values of the control variable for which the loop should continue repeating. From the condition, you should be able to determine the value(s) for which the loop should stop repeating.

3. Changing the value of the control variable

The body of the `while` loop must contain statements that, at some point, will cause the condition of the loop to become False. For example, consider the following code fragment:

```
price = 100.00;
while (price != 0.0)
{
   cout << "Enter a price: ";
   cin >> newPrice;
}
```

Can you predict what will happen when these statements are executed?

Since each new value the user enters is assigned to the variable `newPrice` and not to `price`, `price`'s value remains 100.00 and the condition, `price != 0.0`, remains True. The body of the loop will never stop executing.

Here is another example of a `while` loop that will never end:

```
counter = -1;
while (counter < 0)
   counter--;
```

When you execute a program and it seems as if the computer is not doing anything, there is a good chance that the program contains a never-ending loop. Check the `while` loops in your program to make sure the condition will, at some point, become False.

Subactivity 9.a.iv

In your study notebook, write down a `while` statement for the following logic: While the examination mark entered is not equal to 999, input the next mark.

Subactivity solution

In its most basic form, this loop statement will be:

```
while (examMark != 999)
{
   cout << "Enter the next mark: ";
   cin >> examMark;
}
```

However, following the discussion in point 1 above, we should regard the initialising of the variable `examMark` as part of the `while` structure. If we include that we get:

```
cout << "Enter the first mark: ";
cin >> examMark;
while (examMark != 999)
```

```
{
   cout << "Enter the next mark: ";
   cin >> examMark;
}
```

Subactivity 9.a.v

Write a program that uses a `while` loop to display the numbers from 10 to 20 with a blank line after each number.

Subactivity solution

```
//Displays the numbers 10 to 20 on separate lines
#include <iostream>
using namespace std;

int main( )
{
   int n;

   n = 10; //Initialise n
   while (n <= 20) //Test n
   {
      cout << n << endl << endl;
      n++; //Change n
   }

   return 0;
}
```

Note how we use two `endl`s to ensure the open line between each value.

Subactivity 9.a.vi

In your study notebook, write down what the output of the following code fragment will be (without running it on your computer):

```
a = 10;
cout << "Here's the loop" << endl;
while (a < 10)
   cout << "This is so exciting!" << endl;
```

And what will the output of the following code be?

```
b = 1;
cout << "Here's another loop" << endl;
while (b < 10)
   cout << "b++;" << endl;
```

Subactivity solution

The output of the first code fragment is:

```
Here's the loop
```

The condition `a < 10` is False to begin with, since `a` is assigned the value 10. The body of the loop therefore never executes at all.

The output of the second code fragment is

```
Here's another loop
b++;
b++;
b++;
b++;
b++;
b++;
b++;
b++;
b++;
and so on ...
```

Discussion

The string `"b++;"` is merely a string of characters and does not change the value of `b`. So `b` remains 1 and the condition never becomes False. If you have a never-ending loop like this in your program, the computer will just keep going round and round until you switch it off or reboot it! (Sometimes, if you are lucky, <Ctrl+Break> or <Ctrl+C> will interrupt the loop and stop the program.)

Now that you have worked through these subactivities, you can tackle the problem set for the main activity again. The task is: Write a program that inputs the prices of zero or more items, and displays the total price with 14% VAT added. The program will know when the last price has been input when a price of 0.00 is entered.

Check that your program works correctly for the following test cases:

(i) Three items, costing 12.04, 2.99, and 0.96, followed by 0.00

(ii) Four items: 4.50, 12.20, 2.22, 18.75, followed by 0.00

(iii) No items (i.e. simply enter 0.00)

As you can check with a calculator, the answers to these three should be 18.23, 42.94 and 0.00.

Activity solution

```
//Adds VAT to the total of an unknown number of prices
#include <iostream>
using namespace std;

int main( )
{
   const float VAT_RATE = 0.14;
   float price, total, vat, vatIncl;

   //Initialise price and total
   cout << "Enter the price of the first item: ";
   cin >> price;
   total = price;

   //Input the rest of the prices and add to total
   while (price != 0.0)
   {
      cout << "Enter the next price: ";
      cin >> price;
      total += price;
   }

   //Calculate and display the total including VAT
   vat = total * VAT_RATE;
   vatIncl = total + vat;
   cout << endl << "The total price is " << vatIncl << endl;

   return 0;
}
```

Use the same set of test cases on this program as we suggested a little earlier.

Important points in this lesson

Programming concepts

In this lesson, we have seen how to write a program that does the same task repeatedly, using a `while` loop. We saw that the structure of the `while` loop is

```
while (Condition)
   Statement;
```

The *`Condition`* is a relational expression that can be either True or False, and *`Statement`* can be a single statement or a compound statement. *`Statement`*, which we also call the body of the loop, executes repeatedly, as long as *`Condition`* is True.

The variables that occur in *`Condition`* are called the *control variables*, and are used to control the number of repetitions.

Programming principles

There are three important aspects about the control variable(s) that you should always keep in mind when you use a `while` loop:

1. The control variable (or variables) that appear in the *`Condition`* should always be initialised correctly before entering the `while` loop for the first time.
2. The condition of the `while` loop must test the control variable, specifying the values for which the loop should keep on repeating.
3. The body of the `while` loop must contain a statement which changes the value of the control variable (or variables) so that *`Condition`* becomes False sooner or later. If you do not do this, the program will become stuck in a never-ending loop.

We follow the same indentation conventions with `while` loops as we do with `if` statements. We indent the statement, or block of statements, that form the body of the loop. This helps a reader to see the structure of the `while` loop clearly.

Exercises

Exercise 9.1

What is the least number of times the body of a `while` loop can be executed?

Exercise 9.2

What is wrong with the following program segment? (Hint: Check what the outputs would be for various inputs.)

```
total = 0.0;
cout << "Enter number of values: ";
cin >> num;
count = 0;
while (count < num)
{
   cout << "Enter a value : ";
   cin >> value;
   last = value;
}
cout << "The last value entered was " << last << endl;
```

Exercise 9.3

When Bongi's new baby, Sipho, was born, she opened a savings account with R1000.00. On each birthday, starting with the first, the bank added interest of 4.5% of the balance and Bongi added another R500.00 to the account. Write a loop that will calculate how much money was in the account on Sipho's 18th birthday.

Exercise 9.4

The maximum mass of luggage allowed on a certain kind of airplane is 10 000.00 kg. Write a program that inputs the mass of each item of luggage and decides whether the total mass exceeds the maximum load that the airplane can take.

The program should display an appropriate message to report its finding.

Note that your program will require a `while` loop to calculate the total mass (use 0 as a the final value to indicate the end of input), and then it will need an `if` statement after the loop to determine whether the total mass is greater than the allowed maximum.

Lesson 10

Program debugging

Purpose of this lesson

Developing a program and applying all that we have learnt so far can seem quite easy when someone else does it, but things often do not go so smoothly when trying it yourself! To try to reduce the number of errors during programming, we have emphasised good style and habits such as indenting code and using meaningful variable names to make programs easily readable. All these precautions help a lot, but errors still inevitably occur from time to time. Correcting these errors can be very time-consuming and frustrating, and so in this lesson we are going to look at ways of finding and correcting errors.

Programming errors

Typically, programming errors fall into one of three categories. First there are the *syntax errors*. The compiler checks for syntax errors, and gives error messages that often make it easier to find the error. Programmers become familiar with syntax errors and error messages soon after they start programming! (We looked at syntax errors earlier in these lessons.)

The second type of error is a *run-time error*. This occurs while a program is running and invariably causes the program to crash and a nasty error message to be displayed.

The third type of error is a *logical error*. This occurs when a program gives the wrong results. There are infinitely many types of logical errors. For example, the program does the wrong calculation, or the condition of an `if` statement or a `while` loop is incorrect. A logical error isn't a run-time error, because the program does not crash.

Dealing with logical errors

While syntax and run-time errors are annoying, at least some kind of error message is displayed. Sometimes the error message isn't very helpful, but at least you know there is an error. With logical errors, the computer cannot help you, and very often these are the most difficult kinds of errors to fix. Actually, at first we may not even notice that there has been an error unless we do careful testing. (This is why we emphasise testing so much.)

We are going to work through a typical set of problems with a fictitious programmer called George. This is to help us learn to find errors in programs. (Programmers often call this process "debugging").

The format of this lesson differs from the standard format we have used in the previous lessons. As we go along in this lesson, you will find boxes with exercises. We strongly suggest that you do each of these exercises before going on further. At first you will do them with paper and pencil, and then later on the computer. This is quite an extended lesson, so, instead of doing it all in one go, you may want to break it up into two or three sessions.

George must write a program to read a list of positive integers, and determine the range between them.[2] (In other words, he must subtract the smallest number from the largest, and display the answer.) No number will be greater than 1000.

George does not start programming immediately but first writes a brief specification for this problem.

> Before looking at George's specification, try writing your own. You do not need to follow any specific format.
>
> 
>
> Just jot down the important points about this problem and about how to solve it in your study notebook.

Once you have written your specification, put it to one side and see how George does it. Then you can compare what you have done with what George has done.

Number range program

Inputs: A series of integers between 0 and 1000.

Outputs: The range between the largest and smallest number. Also user prompts, etc.

Process: We need to find the largest number in the series and the smallest, and then to subtract them to find the range.

Finding the largest number: Create a variable called largest. As each number is entered, compare it to largest. If it is bigger, change largest to this new number.

Finding the smallest number: Like finding the largest number! Create a variable called smallest, and if the number entered is smaller, change smallest to this new number. Keep going until all the numbers have been read.

How do we know when we have read in all the numbers? We assume that the user enters 0 when all the numbers have been entered.

Variables: We do not have to store all the values, only the following:

```
largest (for the largest value entered)
smallest (for the smallest value entered)
inValue (for the value currently entered by the user)
range (for largest - smallest)
```

This is what George came up with. How does it compare with what you did? Are there things you thought of that George has not done? Has George included points that had not occurred to you?

George realises that he should probably write out an algorithm for this problem, but he is eager to get his hands on the keyboard, so he starts typing a program into the computer straight away.

> Before looking at what George does, try taking the next step yourself.
>
> 
>
> You may want to write out a clear algorithm if you have not done so already, or you may want simply to write out the program straight away in your study notebook.

[2] This problem is based on an example given in *Turbo Pascal: Programming and Problem Solving* by Leestma and Nyhoff, 1990:156-163.

Here is George's program (we have added line numbers to make it easier to refer to particular lines if we need to):

```
1  //Range of integers
2  #include <iostream>
3  using namespace std;
4
5  int main( )
6  {
7     int largest,   //the largest value entered
8         smallest,  //the smallest value entered
9         inValue,   //the value currently entered
10        range;     //largest - smallest
11
12    cout << "Enter a series of numbers (0 to stop)" << endl;
13
14    //While loop to process all values until user enters 0
15    while (inValue != 0)
16    {
17       cout << "Enter value: ";
18       cin >> inValue;
19       if (inValue > largest)
20          largest = inValue;
21       else
22          if (inValue < smallest)
23             smallest = inValue;
24    } //while loop
25
26    range = largest - smallest;
27    cout << endl << "Range is " << range << endl;
28
29    return 0;
30 }
```

Type in George's program now. We are going to fix it step by step.

Desk checking

George takes a moment to look through his program. He is feeling quite pleased with it, particularly with the `if ... else if ...` part, which seems to solve this problem quite neatly. (Although we have not yet used quite such complicated `if` statements, what George has done is in fact quite legal.)

> Unfortunately for George, he has not done his desk-checking carefully enough. There are some logical errors in this program.
>
> Before going further, can you find these logical errors?

When George runs his program, he gets the following output when he enters 1, 3 and 0:

```
Enter a series of numbers (0 to stop)
```

```
Enter a value: 1
Enter a value: 3
Enter a value: 0
Range is 575
```

Obviously, something is not quite right here! The range should be 2. George tries some other combinations and discovers that the range always seems to be some strange number unrelated to the input.

Can you see what the problem is?

One useful way of diagnosing a problem like this is by looking at the symptoms and then trying to deduce the problem by logical thinking. You might have been able to spot the error this way, but sometimes the output is so strange that logical thinking does not provide much clue as to the problem. In situations where you can't think what the problem can be, variable diagrams can be helpful.

Draw variable diagrams for this program. See if this helps you to pinpoint the problem exactly.

George's variable diagrams look like this:

	largest	*smallest*	*inValue*	*range*
Line 10:	?	?	?	?

	largest	*smallest*	*inValue*	*range*
Line 15:	?	?	?	?

Aha! George has already found a problem. In line 15 of his program `inValue` is undefined. Whatever its value is, it makes the condition of the `while` loop True since he knows that the body of the loop was executed. This is a problem because he can't be sure when the `while` loop will be executed or not.

This may or may not be the reason for the funny output, but it is an error nevertheless. George has come up against the loop initialisation problem mentioned in Lesson 9.

Before looking at the solution George tried, go back to George's program and see if you can correct it. It may help to look back to the **Important points** section of the previous lesson.

The problem is that variable `inValue` is used in the condition of the `while` loop before a value is assigned to it.

George also notices that the other variables `largest` and `smallest` are uninitialised when they are used in lines 19 and 22. To overcome this, George introduces some new statements just after line 10 to initialise the three variables.

```
1 //Range of integers - Version 2
2 #include <iostream>
```

```
3  using namespace std;
4
5  int main( )
6  {
7     int largest,   //the largest value entered
8         smallest,  //the smallest value entered
9         inValue,   //the value currently entered
10        range;     //largest - smallest
11
12    largest = 0;
13    smallest = 1000;
14    inValue = 1;
15
16    cout << "Enter a series of numbers (0 to stop)" << endl;
17
18    //While loop to process all values until user enters 0
19    while (inValue != 0)
20    {
21       cout << "Enter value: ";
22       cin >> inValue;
23       if (inValue > largest)
24          largest = inValue;
25       else
26          if (inValue < smallest)
27             smallest = inValue;
28    } //while loop
29
30    range = largest - smallest;
31    cout << endl << "Range is " << range << endl;
32
33    return 0;
34 }
```

When George runs this program, the output he gets is:

```
Enter a series of numbers (0 to stop):
Enter value: 10
Enter value: 7
Enter value: 15
Enter value: 0
Range is 15
```

This looks a little better, but the result is still wrong. The range should be (15 - 7) = 8, but instead the program shows the range as 15.

> Take some time now to check this program and see whether you can find out why it gives the wrong answer.

If you cannot find a fault like this just by working through the program, a good way to find it is to use our trusty old variable diagrams again:

	largest	smallest	inValue	range
Line 14:	0	1000	1	?

	largest	smallest	inValue	range
Line 19:	0	1000	1	?

	largest	smallest	inValue	range
Line 22:	0	1000	10	?

	largest	smallest	inValue	range
Line 23+24:	10	1000	10	?

	largest	smallest	inValue	range
Line 19:	10	1000	10	?

	largest	smallest	inValue	range
Line 22:	10	1000	7	?

	largest	smallest	inValue	range
Line 26+27:	10	7	7	?

	largest	smallest	inValue	range
Line 19:	10	7	7	?

	largest	smallest	inValue	range
Line 22:	10	7	15	?

	largest	smallest	inValue	range
Line 23+24:	15	7	15	?

	largest	smallest	inValue	range
Line 19:	15	7	15	?

	largest	smallest	inValue	range
Line 22:	15	7	0	?

	largest	smallest	inValue	range
Line 26+27:	15	0	0	?

Here we can see the problem. The value of `smallest` changes from 7 to 0. This 0 was entered to signal the end of the input data, and is not a valid data value.

An alternative debugging method

We have just pinpointed the problem by working through the program manually using variable diagrams. Before trying to fix this problem, let us look at another way of trying to find this same error.

Instead of working out the value of each variable by hand at each step (and drawing a variable diagram), we can get the computer to display these values by inserting temporary output statements into the program to display these values.

Try doing this now for yourself.

After line 22, insert the statement

```
cout << "inValue: " << inValue << endl;
```

to echo the input value.

At the end of the `while` loop, just before the } in line 28, insert the statements

```
cout << "largest: " << largest;
cout << ", smallest: " << smallest << endl << endl;
```

The `while` loop should now look like this:

```
19    while (inValue != 0)
20    {
21       cout << "Enter value: ";
22       cin >> inValue;
23       cout << "inValue: " << inValue << endl;
24       if (inValue > largest)
25          largest = inValue;
26       else
27          if (inValue < smallest)
28             smallest = inValue;
29       cout << "largest: " << largest;
30       cout << ", smallest: " << smallest << endl << endl;
31    } //while loop
```

Running the program with these temporary `cout` statements and the same input values gives:

```
Enter a series of numbers (0 to stop):
Enter value: 10
inValue: 10
largest: 10, smallest: 1000

Enter value: 7
inValue: 7
largest: 10, smallest: 7

Enter value: 15
inValue: 15
largest: 15, smallest: 7

Enter value: 0
```

```
inValue: 0
largest: 15, smallest: 0

Range is 15
```

Compare this with the variable diagrams, and you will see the same pattern emerges in both. All goes well except the first value of `smallest`, and until the user enters the final 0 (zero). This sets `smallest` to 0 and so gives the wrong range.

We have now found out what the error is in two different ways.

Can you fix it?

Once you have changed the program, run it with the same test data to see whether it now works properly.

George realises that to solve this problem he must sit down and think quite carefully. The difficulty is that after inputting the zero, which means "Stop processing", the program does not stop processing, but instead treats the 0 as valid input.

There are two ways George can fix his problem. The first way is that, after inputting a value, the program can test this value with an `if` statement. It will only be processed if it is greater than zero. This new version of the `while` loop would be:

```
19    while (inValue != 0)
20    {
21       cout << "Enter value: ";
22       cin >> inValue;
23       if (inValue > 0)
24          if (inValue > largest)
25             largest = inValue;
26          else
27             if (inValue < smallest)
28                smallest = inValue;
29    } // while loop
```

But George does not feel happy with this - it just seems too complicated. He gets out a piece of paper, and tries to write down a simple algorithm for the problem:

- *Input an item of data;*
- *If it is valid, process it and go on to the next item of data;*
- *If it is not valid, stop processing the input data and go on to the next step.*

How does one write this in C++? Well, since we must repeat a series of actions an unknown number of times, we should rather use the `while` statement than the `if` statement shown above.

```
Input an item of data
while the data is not the end-of-data signal,
   Process the input data
   Input the next item of data
```

Converting this into a program gives:

```
1  //Range of integers - Version 4
2  #include <iostream>
3  using namespace std;
4
5  int main( )
6  {
7     int largest,         //the largest value entered
8         smallest,         //the smallest value entered
9         inValue,        //the value currently entered
10        range;         //largest - smallest
11
12    largest = 0;
13    smallest = 1000;
14
15    cout << "Enter a series of numbers (0 to stop)" << endl;
16    cout << "Enter value: ";
17    cin >> inValue;
18
19    //While loop processes all values until user enters 0
20    while (inValue != 0)
21    {
22       if (inValue > largest)
23          largest = inValue;
24       else
25          if (inValue < smallest)
26             smallest = inValue;
27       cout << "Enter value: ";
28       cin >> inValue;
29    } //while loop
30
31    range = largest - smallest;
32
33    cout << endl << "Range is " << range << endl;
34
35    return 0;
36 }
```

Compare this new version of the program to the simple algorithm we gave just before it, and make sure that you understand how these two connect.

In comparison to version 2 of the program, we input an initial value for `inValue` rather than initialising it to 1. We have also changed the order of the statements in the `while` loop to make sure that we do not treat 0 as a valid input.

When George runs his new program, the output he gets is:

```
Enter a series of numbers (0 to stop):
Enter value: 10
Enter value: 7
Enter value: 15
Enter value: 0
```

```
Range is 8
```

At last - the right answer! George is beginning to feel quite pleased with himself. Just to be sure, though, George begins testing it with several different combinations of numbers. He finds that it gives a very strange answer when each number is larger than the previous one:

```
Enter a series of numbers (0 to stop):
Enter value: 7
Enter value: 10
Enter value: 15
Enter value: 0

Range is  985
```

It also gives the wrong range if he enters the numbers in the order 7, 15, 10. (Try this for yourself.)

> Take one last look at this program to see if you can find the remaining error.
>
> You can try to find it by inspection (i.e. just by reading through the program code)
>
> If this does not work, try either using variable diagrams or inserting temporary `cout` statements to inspect the values of important variables.

George decides temporarily to display the values of `inValue`, `largest`, and `smallest` just after the compound `if` statement:

```
19    //While loop processes all values until user enters 0
20    while (inValue != 0)
21    {
22       if (inValue > largest)
23          largest = inValue;
24       else
25          if (inValue < smallest)
26             smallest = inValue;
27       cout << "inValue: " << inValue;
28       cout << ", largest: " << largest;
29       cout << ", smallest: " << smallest << endl << endl;
30       cout << "Enter value: ";
31       cin >> inValue;
32    } //while loop
```

Running this gives:

```
Enter a series of numbers (0 to stop):
Enter value: 7
inValue: 7, largest: 7, smallest: 1000

Enter value: 10
inValue: 10, largest: 10, smallest: 1000
```

```
Enter value: 15
inValue: 15, largest: 15, smallest: 1000

Enter value: 0

Range is  985
```

We see that `smallest` always stays at the value it was initialised to. This means that, with this series of values, the `else if` part in the body of the loop never executes - can you see why?

In this case, each succeeding value is larger than the previous one. The condition of the `if` statement is always True, and so the `else` part never executes.

There are two ways we can change this program to give the correct answer. Instead of using the complicated `if .. else if ..` statement to test `inValue` against either `largest` or `smallest` (but not both), we can test `inValue` against both `largest` and `smallest` by using two consecutive `if` statements:

```
19    //While loop processes all values until user enters 0
20    while (inValue != 0)
21    {
22       if (inValue > largest)
23          largest = inValue;
24       if (inValue < smallest)
25          smallest = inValue;
26       cout << "inValue: " << inValue;
27       cout << ", largest: " << largest;
28       cout << ", smallest: " << smallest << endl << endl;
29       cout << "Enter value: ";
30       cin >> inValue;
31    } //while loop
```

Alternatively, by changing the order of the statements, we can initialise `largest` and `smallest` to the first `inValue` that we input, i.e. before the `while` loop:

```
12    cout << "Enter a series of numbers (0 to stop)" << endl;
13    cout << "Enter value: ";
14    cin >> inValue;
15    smallest = inValue;   // better initialisation
16    largest = inValue;    // better initialisation
```

Make one of these sets of changes, and then test the program thoroughly. The tests we did above show that the order of the input values can make a difference to the program behaviour. So, one test you should try on the revised program is to take the same three numbers we have used up until now, and try them in all the different orders they can be in. (This gives six test cases: 7, 10, 15; 7, 15, 10; 10, 15, 7; 10, 7, 15; 15, 7, 10; 15, 10, 7.) Does the program handle all of these correctly?

And what happens if the user inputs values outside of the 0 ... 1000 range that we designed the program for? Test the program with inputs like: 7, 15, 1020; 15, 7, -3.

A simple way to make sure that the program rejects negative values and values above 1000, is to modify the condition in the `while` loop:

```
19    while (inValue > 0 && inValue < 1001)
```

Make this change, and test it again. (We will see what `&&` means in the next lesson.) Try a few other input combinations. What happens if you enter 0 straight away? What happens if you enter the same number several times?

We tried all these different combinations of inputs, and the program gave the right answer each time. Finally, we seem to have a program that works under all the input conditions we can think of!

Important points in this lesson

Programming concepts

We saw that we can use an `if` statement inside another `if` statement. This is explained in more detail in Lesson 12.

Programming principles

The computer tells you when there are syntax or run-time errors in a program. However, you (the programmer) have to find the logical errors. As programs become more complex, logical errors can become more difficult to find. Before typing in a program, it is a good idea to work out on paper first what exactly the program should do, and to sketch out an algorithm that will do this.

It is important always to test a program with a variety of test data. If the program gives wrong answers, you can try to track down the error through inspection, by drawing variable diagrams, or by inserting temporary `cout` statements. One may have to go through several cycles of testing and correcting the program before it works completely satisfactorily.

To end this lesson, we can make some general comments about the programming process:

- There is almost always more than one correct solution to a programming problem.
- You should always be ready to revise your ideas when the program is not working well.
- It is often very difficult to get a program right the first time.

Lesson 11

Boolean values

Purpose of this lesson

The conditions of `if` and `while` statements that we have seen so far contain the relational operators `==`, `<`, `>` etc. These conditions give boolean values, i.e. True or False.

To perform operations on boolean values, C++ provides three operators `&&`, `||` and `!`, representing logical AND, OR and NOT respectively. We use them to create compound conditions for `if` and `while` statements.

C++ also provides a boolean type called `bool` that we can use to declare boolean variables (i.e. variables that can store the values `true` or `false`) and to perform operations on boolean values.

Activity 11.a

Write a program that asks the user for an integer between 10 and 20. It must then test whether the number is indeed between 10 and 20, and if not, repeatedly ask for the number to be entered again until it is within the correct boundaries.

Test yourself

You should realise that you need a `while` loop for this program. You might think that two `if` statements, or an `if` statement within a `while` is the answer, but fortunately it's easier than that.

The secret is the condition of the `while` loop. We gave a clue in the paragraph stating the purpose of this lesson above.

If you are not too sure how to proceed, take a look at the following subactivities.

Subactivity 11.a.i

Write a short program that inputs a value into an integer variable `i`, and then outputs the result of the relational operation `i < 10`. Do this with the statement

```
cout << "i < 10 is " << (i < 10) << endl;
```

Subactivity solution

Surprised? We said earlier that the result of a relational operation is a boolean value. Although this is true, this subactivity shows that the boolean values `true` and `false` are actually equivalent to 1 and 0 respectively.

The following subactivity illustrates the same thing, but in another way.

Subactivity 11.a.ii

Write a short program (or rather edit the one you wrote for the previous subactivity) that includes the following `if` statement:

```
if (i)
   cout << "true" << endl;
else
   cout << "false" << endl;
```

using an integer variable `i` whose value is input from the keyboard.

Run the program a few times and experiment with different values of `i` to see when `true` and `false` are displayed.

Subactivity solution

Confused? This subactivity shows that C++ treats any non-zero integer as `true` and 0 as `false`.

Discussion

We consider it poor programming style to write an `if` statement with an integer value (or expression) for its condition like this. A relational operator should rather be used. In other words, we prefer to write the above `if` statement as

```
if (i != 0)
   cout << "true" << endl;
else
   cout << "false" << endl;
```

so that we can interpret the condition as `true` or `false`.

Subactivity 11.a.iii

Type in the following program, compile and run it:

```
//Using a boolean operator
#include <iostream>
using namespace std;
```

```
int main( )
{
   int n;

   cout << "Enter an integer between 10 and 20: ";
   cin >> n;

   if (n > 10 && n < 20)
      cout << "Thank you!" << endl;
   else
      cout << n << " is not between 10 and 20!" << endl;

   return 0;
}
```

Now change the condition of the `if` statement to `(10 < n < 20)` and run it again. What is the difference?

Subactivity solution

The program works correctly the first way, but incorrectly the second.

Discussion

Here we see the use of the boolean operator `&&` representing logical AND.

You should read the `if` statement

```
if (n > 10 && n < 20)
```

as "if `n` is greater than 10 AND `n` is less than 20".

Note that you should not write

```
if (10 < n < 20)
```

The compiler will accept this, but the program will give the wrong answer. (This is an example of a logical error.)

To see why, consider the situation where `n` is 30. The computer will first evaluate `10 < n` which will be `true`. Then the computer tries to evaluate `true < 20`. Remember that `true` is equivalent to 1, so `true < 20` is equivalent to `1 < 20` which is `true`. So the whole condition will evaluate to `true`. This is wrong because 30 is not between 10 and 20.

Beware of this mistake!

You should now be able to write the program for the main activity. There are many ways to write the condition of the `while` loop. You can use the logical OR operator `||` or you can use `&&` and `!` (representing logical AND and NOT respectively).

Activity solution

```
//Checking for valid input
#include <iostream>
using namespace std;

int main( )
{
   int n;

   cout << "Enter an integer between 10 and 20: ";
   cin >> n;

   while (n <= 10 || n >= 20)
   {
      cout << n << " is not between 10 and 20!" << endl;
      cout << "Enter again: ";
      cin >> n;
   }
   cout << "Thank you!" << endl;

   return 0;
}
```

An alternative condition for the `while` loop would be `(!(n > 10 && n < 20))`

Activity 11.b

Write a program to input two integers and to determine if the first number is a factor of the second. (One number is a factor of another if it divides into the other number without a fraction.) Here is an example of the output of the program:

```
Enter two integers: 4 12
4 is a factor of 12
```

Test yourself

This is really an easy program to write. There are a number of interesting issues that it raises, however.

If you have written a program for this problem, before taking a look at our solution, test your program with negative numbers, and also make sure that it works for zero.

The first subactivity helps you to write the simplest version of the program. Subsequent subactivities explain how to improve it to deal with all possible inputs.

Subactivity 11.b.i

Here is the outline of the program for the solution to the main activity:

- Input two integers into variables.
- Use an `if` statement to test whether the first number is a factor of the second. Remember that if one number is a factor of another, the remainder when the second is divided by the first will be zero. In other words, use the `%` operator in the condition of the `if` statement.
- Display appropriate messages for the two possibilities.

Write a program that implements these steps.

Subactivity solution

```
//Checks whether one number is a factor of another
#include <iostream>
using namespace std;

int main( )
{
   int x, y;
   cout << "Enter two integers: ";
   cin >> x >> y;

   if (y % x == 0)
      cout << x << " is a factor of " << y << endl;
   else
      cout << x << " is not a factor of " << y << endl;

   return 0;
}
```

Discussion

Test the above program for positive as well as negative numbers, and confirm that it works correctly for them all.

Then test the program using the value zero. It should work correctly if the second value is zero, but the program crashes when the first value is zero. The computer can't divide `y` by 0. This is an example of a run-time error.

Subactivity 11.b.ii

Fix the above program to work properly even if zero is entered for the first number. Nest the current `if` statement in another `if` statement that first tests whether `x != 0`. Note that 0 is not a factor of `y`, no matter what the value of `y` is.

Subactivity solution

```
//Checks whether one number is a factor of another
#include <iostream>
using namespace std;

int main( )
{
   int x, y;
   cout << "Enter two integers: ";
   cin >> x >> y;

   if (x != 0)
      if (y % x == 0)
         cout << x << " is a factor of " << y << endl;
      else
         cout << x << " is not a factor of " << y << endl;
   else
      cout << x << " is not a factor of " << y << endl;

   return 0;
}
```

Subactivity 11.b.iii

We can simplify the above program by using a single `if` statement with a compound condition, namely `x != 0 && y % x == 0`. Make this change to get the final solution to the main activity.

Activity solution

```
//Checks whether one number is a factor of another
#include <iostream>
using namespace std;

int main( )
{

   int x, y;
   cout << "Enter two integers: ";
   cin >> x >> y;

   if (x != 0 && y % x == 0)
      cout << x << " is a factor of " << y << endl;
   else
      cout << x << " is not a factor of " << y << endl;

   return 0;
}
```

Discussion

We prefer this solution because it is shorter and neater than the previous one. It illustrates the fact that nested `if` statements can often be simplified to a single `if` statement by combining their conditions into a compound condition.

We assume that you have tested the above program for all combinations of values. In particular, it works when `x` is 0, which seems to suggest that the expression `y % x` is not executed at all.

You can check this by swapping the two parts of the compound condition of the `if` statement. In other words, use the condition `y % x == 0 && x != 0`. You will see that the program crashes.

This shows that C++ uses so-called *short-circuit boolean evaluation* when the boolean operator `&&` is used. Short-circuit boolean evaluation works as follows: If the first operand of `&&` evaluates to `false`, the second operand is not evaluated.

The reason for this short-cut is that if the first operand of `&&` is `false`, the whole boolean operation will end up being `false`, no matter what the second operand is. In this case, C++ ignores the second operand for the sake of efficiency.

In fact, a similar thing happens with the `||` operator. If the first operand of `||` is evaluated as `true`, the second operand is not evaluated. The same argument applies: If the first operand is `true`, the whole expression will be `true`, no matter what the second operand is.

Activity 11.c

Consider the following program for testing whether a number is a prime number:

```
//Test whether a number is prime
#include <iostream>
using namespace std;

int main( )
{
   int x, y;
   cout << "Enter a positive integer: ";
   cin >> y;
   x = 2;
   while (x != y)
   {
      //test if x is a factor of y
      if (y % x == 0)
         cout << y << " is not prime" << endl;
      x++;
   }
   if (x == y)
      cout << y << " is prime!" << endl;

   return 0;
}
```

This program works correctly if a prime number is entered. However, if a non-prime number like 12 is entered, the message `12 is not prime` is displayed over and over, as well as the message `12 is prime`.

Fix these problems and hence make the above program more efficient by introducing a boolean variable.

Test yourself

Could you improve the program? You probably first need to know what a prime number is. Then you need to convince yourself that the program continues unnecessarily after having determined that the number is not prime and is not as efficient as it could be. Finally you need to know how to introduce a boolean variable.

Before we look at each of these issues in turn, note that we don't have to protect the expression `y % x == 0` from causing the program to crash, as we did in the previous activity, because x can't be zero. It starts at 2, and is incremented inside the loop.

> A number is a prime number if it only has two factors, namely 1 and itself. (We saw what a factor is in the previous activity. If you've forgotten, you should turn back and look it up.) For example, 12 is not a prime number since its factors are 1, 2, 3, 4, 6 and 12 itself. 13 is a prime number, since the only factors it has are 1 and 13 itself.

The first subactivity gets you to think about the repeated message and the question of efficiency. The next few subactivies explain how to declare and use a boolean variable in a program. The final two subactivities help you to incorporate these ideas into the program to form the solution to the activity.

Subactivity 11.c.i

Explain why the same message is displayed if the number is not prime. (If you aren't convinced that it does this, type in the program and test it out with a number like 12.)

Subactivity solution

The answer is that if a number has a factor, then it probably has another factor, which when multiplied together give the number. For example, say we are testing the number 12. When 2 is reached, the message is displayed (because 2 * 6 is 12). Later, when 6 is reached, the message will be displayed again. The same is true for 3 and 4.

The problem lies in the fact that the `while` loop keeps on executing after a factor is found. Think about it. As soon as a factor is found, the loop can terminate. If we can fix this, the program will be more efficient as well, because it will terminate sooner.

Discussion

The main activity required you to introduce a boolean variable into the program to make it more efficient.

When a `while` loop must be terminated prematurely, it is a common technique to use a boolean variable as a so-called "flag". The condition of the `while` loop is turned into a compound condition including the flag as one of its boolean expressions. Then the flag is set inside the body of the `while` loop if it must be terminated sooner than is expected.

But enough waffling! You first need to know how to introduce a boolean variable:

Subactivity 11.c.ii

Type the following program into the computer, compile and run it to check what it does.

```
//Using a boolean variable
#include <iostream>
using namespace std;

int main( )
{
   bool result;
   int n1, n2;

   cout << "Enter two integers: ";
   cin >> n1 >> n2;

   result = n1 > n2;
   if (result)
      cout << n1 << " is greater than " << n2 << endl;
   else
      cout << n1 << " is not greater than " << n2 << endl;

   return 0;
}
```

Now edit the program so that it doesn't use a boolean variable.

Subactivity solution

```
//Without using a boolean variable
#include <iostream>
using namespace std;

int main( )
{
   int n1, n2;

   cout << "Enter two integers: ";
   cin >> n1 >> n2;

   if (n1 > n2)
      cout << n1 << " is greater than " << n2 << endl;
   else
```

```
      cout << n1 << " is not greater than " << n2 << endl;

   return 0;
}
```

That was easy enough.

Discussion

We can declare a boolean variable just like any other variable. We can also assign a value to a boolean variable by assigning it the result of a relational operation (two values on either side of a relational operator).

As you can see, we can very often get by without a boolean variable, but sometimes they are useful, as we will see later.

Subactivity 11.c.iii

Edit the program for Subactivity 11.a.iii to use a boolean variable.

Subactivity solution

```
//Using a boolean variable and a boolean operator
#include <iostream>
using namespace std;

int main( )
{
   bool result;
   int n;

   cout << "Enter an integer between 10 and 20: ";
   cin >> n;

   result = n > 10 && n < 20;
   if (result)
      cout << "Thank you!" << endl;
   else
      cout << n << " is not between 10 and 20!" << endl;

   return 0;
}
```

Easy-capeasy!

Subactivity 11.c.iv

You are now ready to incorporate a boolean variable into the program of the main activity. For this, you will need to declare a boolean variable, say `factorFound` at the beginning of the program and initialise it to `false`. As stated in the **Discussion** section after the first subactivity, you need to include this boolean variable (called the flag) in the condition of the `while` loop.

Write down the compound condition of the `while` loop including the boolean variable.

Subactivity solution

```
while (x != y && !factorFound)
```

Did you put the NOT symbol (`!`) in? If so, well done!

Discussion

You can check whether you need to use it or not by "reading" the `while` loop as follows: "Repeat the loop while `x` is NOT equal to `y` AND factor is NOT found." Of course, reading a `while` loop in this way will only work if you have given the boolean variable a meaningful name and have initialised it correctly.

Subactivity 11.c.v

You are now ready to put the finishing touches to the program.

All you need to do is test whether `x` is a factor of `y` in the body of the loop and set `factorFound` to `true` if it is.

Although not essential, there is one more change that you can make: Instead of displaying the message `x is not prime` inside the loop, you can now move it outside (after) the loop, namely to the `else` part of the second `if` statement. We prefer it, because these two messages belong together.

Activity solution

```
//Test whether a number is prime
#include <iostream>
using namespace std;

int main( )
{
   int x, y;

   bool factorFound = false;

   cout << "Enter a positive integer: ";
   cin >> y;
   x = 2;
   while (x != y && !factorFound)
   {
      if (y % x == 0)
         factorFound = true;
      x++;
   }
   if (x == y)
      cout << y << " is prime!" << endl;
```

```
   else
      cout << y << " is not prime." << endl;

   return 0;
}
```

Well done if you did all the subactivities and worked out the final solution! You'll see other interesting things you can do with boolean variables in the next activity.

Activity 11.d

This isn't an activity at all but rather a number of subactivities that illustrate some other interesting features of `if` statements and boolean variables.

Subactivity 11.d.i

The Department of Security Services wants to identify single young people with ages from 18 to 26 years who have passed matric. Using two boolean variables `single` and `matric`, and an integer variable `age`, write an `if` statement to determine whether a person is a suitable candidate and display an appropriate message.

Subactivity solution

```
if (single && matric && age >= 18 && age <= 26)
   cout << "Good candidate" << endl;
else
   cout << "Not suitable" << endl;
```

Subactivity 11.d.ii

Rewrite the following two `if` statements so that no `if` appears in them:

(i)
```
if (age >= 18)
   mayVote = true;
else
   mayVote = false;
```

(ii)
```
if (length > 1.8 && weight < 70)
   overWeight = false;
else
   overWeight = true;
```

Subactivity solution

(i)
```
mayVote = age >= 18;
```

(ii)
```
overWeight = !(length > 1.8 && weight < 70);
```

Discussion

Any `if..else` statement which has the sole purpose of assigning the value `true` or `false` to a boolean variable, can be rewritten as a single assignment statement. Instead of testing the condition (a boolean expression) to decide whether `true` or `false` should be stored in the relevant boolean variable, the boolean expression can be assigned directly to the boolean variable.

Subactivity 11.d.iii

Write a single assignment statement to do the same test as in Subactivity 11.d.i. Instead of displaying a message, the statement must assign the value `true` to a variable called `goodCandidate` if the person is a good candidate, and `false` otherwise.

Subactivity solution

```
goodCandidate = single && matric && age >= 18 && age <= 26;
```

Important points in this lesson

Programming concepts

C++ provides two boolean values, namely `true` and `false`. They are generally used to determine the value of a condition of an `if` statement or a loop.

Internally, `true` and `false` are stored as 1 and 0 respectively.

One can in fact provide an integer value wherever a boolean value is required. In this case, all non-zero values are treated as `true`, and 0 is treated as `false`.

The result of a relational operation (i.e. an expression involving one of the relational operators `==`, `<`, `>` etc.) is a boolean value.

The following boolean operators are available: `&&`, `||` and `!`, representing logical AND, OR and NOT respectively. Their operands are boolean values, and the result of the operation is a boolean value.

C++ uses short-circuit boolean evaluation. This means that if `&&` or `||` are used in a boolean expression, the whole expression might not be evaluated. In particular, if the first operand of `&&` evaluates to `false`, the second operand is ignored and the whole expression is evaluated to `false`. If the first operand of `||` evaluates to `true`, the second operand is ignored and the whole expression is evaluated to `true`.

We can declare boolean variables (of type `bool`), and we can assign values to them in assignment statements. We can use boolean variables in boolean expressions involving the boolean operators `&&`, `||` and `!` together with other boolean variables or boolean expressions provided by relational operations.

Programming principles

It is sometimes useful to introduce a boolean variable in a program to provide the condition (or part of a compound condition) of an `if` or `while` statement.

An `if..else` statement which simply assigns the value `true` or `false` to a boolean variable can always be rewritten as a single assignment statement. Consider the following examples:

if..else	Equivalent assignment statement

`if (x >= 0)`
`  nonNegative = true;`
`else`
`  nonNegative = false;`
	`nonNegative = x >= 0;`

`if (rainy || windy)`
`  swimming = false;`
`else`
`  swimming = true;`
	`swimming = !(rainy || windy);`

It is better to use a single assignment statement like these examples rather than an `if` statement with two assignment statements.

It is poor programming practice to use an integer value for the condition of an `if` or `while` statement.

Exercises

Exercise 11.1

The municipality of a small town needs a program to calculate the amount payable for water consumption for each home (or business). The rate depends on the number of units of water that were used.

- The first 20 units are free.
- A fixed rate of R10 per unit is payable for the additional units if 40 units or less are used.
- If more than 40 units but not more that 100 units are used, the cost is 1.5 times the fixed rate (for the additional units).
- If more than 100 units are used, the cost is 2 times the fixed rate (for the additional units).

Write a program to input the units used (as a floating point number) and to output the amount payable.

Exercise 11.2

There are a number of criteria for a toddler being accepted at a particular playschool: (i) The child must be 3, 4 or 5 years old. (ii) The parent must be single. (iii) The parent's annual income must be less than R60 000. (iv) The parent must be 30 years or younger. Write a program to assess whether the criteria are met.

The program should contain a *single* `if` statement that uses boolean variables only for its condition.

Exercise 11.3

Rewrite the following `if` statements as single assignment statements:

(i)
```
if (grade > 7)
   highSchool = true;
else
   highSchool = false;
```

(ii)
```
if (age < 13 || age > 19)
```

```
        teenager = false;
    else
        teenager = true;
```

(iii)
```
if (x < 0)
    found = false;
else if (x % 4 == 0)
    found = true;
else
    found = false;
```

Exercise 11.4

Write a program to give the user 10 chances to guess a number between 1 and 100.

If the user guesses the number correctly in 10 or less tries (say 7), the program should display the message:

```
Well done! You got the number in 7 guesses.
```

Otherwise, the program should display the message:

```
Tough luck! Your 10 chances are over.
```

The program must contain a constant called `SECRET` (of type integer) to hold the number that the user must try to guess. (You can choose any value for `SECRET`, say 23.)

Furthermore, the program must use a boolean variable called `found` which is set to `true` as soon as the user guesses the number correctly.

Lesson 12

Nested if statements

Purpose of this lesson

In this lesson we discuss using nested `if` statements, i.e. `if` statements within other `if` statements (as George did in his program in Lesson 10). Since program code which contains nested `if` statements is often confusing to read, we place emphasis on the layout of such `if` statements.

Activity 12.a

A certain restaurant pays its waitrons (waiters and waitresses) by the hour. They are paid R32.50 an hour in the afternoon (between 1 and 6 o'clock) and R44 in the evening (between 6 and 12 o'clock). Shifts always change on the hour.

Write a program that reads in the starting and finishing times of a waitron and calculates the wage for the work done.

An example of the user dialogue would be:

```
Wage calculation
================
Starting time: 5
Finishing time: 8

The payment is R 120.50
```

Test yourself

For this program you need nested `if` statements, i.e. `if` statements within other `if` statements. Although it is quite tricky, you might like to try to write this program before you tackle the subactivities below. If you can write the program without any problem, you can jump to Activity 12.b. If you feel unsure about your answer or don't know where to start, work through the following subactivities.

Background

As revision, read through Lesson 8 where `if` statements are discussed, again.

We can summarise the structure of a simple `if` statement as follows

```
if (Condition)
   Statement;
```

An `if..else` statement looks like this:

```
if (Condition)
   Statement1;
else
   Statement2;
```

Statement, *Statement1* and *Statement2* can also be compound statements, i.e. a number of statements placed between braces.

If *Condition* is `true`, the statement directly after the condition is executed. For a simple `if` statement, if *Condition* is `false`, the statement following the entire `if` statement in the program is executed, and for an `if..else` statement, the statement in the `else` part is executed.

Subactivity 12.a.i

Write an `if` statement (or statements) to increment (i.e. add 1 to) one of three variables, `numPos`, `numNeg` and `numZero`, depending on the value of a variable `x`. If `x` is greater than 0, add 1 to `numPos`, if `x` is less than 0, add 1 to `numNeg`, otherwise add 1 to `numZero`.

Subactivity solution

There is more than one way to answer this question. One solution is:

```
if (x > 0)
   numPos = numPos + 1;
if (x < 0)
   numNeg = numNeg + 1;
if (x == 0)
   numZero = numZero + 1;
```

(You could also use the compound assignment operators `+=` or `++`.)

Here we have used a sequence of three separate `if` statements. It is possible to do it with only two `if` statements, as given in the solution below. If you only used two `if` statements for your solution, well done!. Although the above solution will work correctly, it is not a very efficient way of using `if` statements. If the value of `x` is 10, the condition of the first `if` statement will be `true` and the value of `numPos` will therefore be adjusted. Then the computer will still test the conditions of the remaining two `if` statements and determine that both are `false`. By using nested `if` statements, we can avoid any unnecessary work. Consider this solution:

```
if (x > 0)
   numPos = numPos + 1;
else
   if (x < 0)
      numNeg = numNeg + 1;
   else
      numZero = numZero + 1;
```

The first `if` statement now has an `else` part. If the condition `x > 0` is `true`, then the value of `numPos` will be incremented and execution of the `if` statements will be completed. However, if the condition is `false`, the `else` part will be executed. Now there is a nested `if` statement in this part. The condition `x < 0` is tested to determine which one of `numNeg` or `numZero` to increment.

Note how indentation has been used in the `if` statement above. When the layout of an `if` statement is not done carefully, it is often difficult to determine what it does. For example, consider the following code:

```
if (x > 0)
numPos = numPos + 1;
else if (x < 0)
numNeg = numNeg + 1;
else numZero = numZero + 1;
```

Although it is identical to the previous code, it is a lot more difficult to follow. The following format is also confusing:

```
if (x > 0)
   numPos = numPos + 1;
   else
   if (x < 0)
      numNeg = numNeg + 1;
else
   numZero = numZero + 1;
```

Once again, the code is identical to the previous code. However, if you look at it quickly, you could mistakenly think that the second `else` belongs to the first `if`. By using suitable indentation, we can improve the readability of an `if` statement considerably.

Note, however, that indentation has no significance for the C++ compiler. All three forms of the nested `if` statement above are interpreted in precisely the same way by the compiler.

Subactivity 12.a.ii

The `if` statements below were written to implement the following logic: If the value of `x` is positive, it must be decreased by 1. If it is negative, it must be increased by 1. However, if `x` is equal to 0, the value of variable `y` must be assigned to it.

```
if (x > 0)
   x = x - 1;
if (x < 0)
   x = x + 1;
if (x == 0)
   x = y;
```

Apart from the criticism that a sequence of separate `if` statements is usually less efficient than nested `if` statements, there is a more serious problem with this code. Identify the problem and correct it.

Subactivity solution

The problem is that if the original value of `x` is 1 or -1, one of the first two `if` statements will change the value of `x` to 0. This causes the condition of the third `if` statement to be `true`, and `x` is assigned the value of `y`. This does not correspond to the requirement that `x` should be assigned the value of `y` only if the original value of `x` is 0.

The only way to deal with this situation is to use nested `if` statements. Here is one solution:

```
if (x > 0)
   x = x - 1;
else
   if (x < 0)
      x = x + 1;
   else
      x = y;
```

Test this solution with a few values to see that it works correctly.

Here is another solution:

```
if (x == 0)
   x = y;
else
   if (x > 0)
      x = x - 1;
   else
      x = x + 1;
```

Test this solution with a few values.

Did you test what happens if `y` is positive, negative or 0? Check whether this will not affect the changes made to `x` in the inner `if` statement.

Discussion

In the first subactivity, we saw how nested `if` statements can be used in place of a sequence of separate `if` statements to improve the efficiency of a program. A sequence of separate `if` statements always requires the condition of each `if` statement to be tested, whereas unnecessary testing of conditions can be avoided by means of nested `if` statements.

In the above subactivity however, we have seen that nested `if` statements are sometimes essential, i.e. a sequence of `if` statements will not do the job properly. This is specifically the case when the later `if` statements in a sequence of `if` statements test a variable whose value is (possibly) changed by the previous `if` statements. Nested `if` statements can be used to completely avoid this problem, since only one statement (or compound statement) of a nested `if` statement is ever executed.

The other lesson to be learnt from the subactivity above is that it is always a good idea to test the logical correctness of your code with a number of values, to make sure that it does indeed do what you want it to.

Subactivity 12.a.iii

Now try and write a nested `if` statement for the problem in the main activity. It must calculate the wage for a number of hours of work from a starting time to a finishing time.

The code can assume that two constants have been declared, namely `AFTERNOON_RATE` and `EVENING_RATE`.

Hint: There are three situations to consider: Both the starting and finishing times are before six o'clock; the starting time is before 6 o'clock but the finishing time is after 6 o'clock; and both the starting and finishing times are after 6 o'clock.

The calculation of the wage for the first situation would be

```
wage = (finish - start) * AFTERNOON_RATE;
```

Subactivity solution

As in the first subactivity we can either tackle this with a sequence of separate `if` statements or with nested `if` statements. Once again, we prefer nested `if` statements because they are more efficient.

Here is one solution using a sequence of separate if statements:

```
if (finish <= 6)
   wage = (finish - start) * AFTERNOON_RATE;
if (start < 6 && finish > 6)
   wage = (6 - start) * AFTERNOON_RATE + (finish - 6) * EVENING_RATE;
if (start >= 6)
   wage = (finish - start) * EVENING_RATE;
```

Here is another solution using nested `if` statements:

```
if (start < 6)
   if (finish <= 6)
      wage = (finish - start) * AFTERNOON_RATE;
   else
      wage = (6 - start) * AFTERNOON_RATE + (finish - 6) * EVENING_RATE;
else
   wage = (finish - start) * EVENING_RATE;
```

Subactivity 12.a.iv

Write the rest of the program for the main activity. (No further `if` statements are necessary.)

Activity solution

```
//Calculates wage for waitron given starting and finishing times
#include <iostream>
using namespace std;
```

```
int main( )
{
   const float AFTERNOON_RATE = 32.50;
   const float EVENING_RATE = 44.00;
   int start, finish;
   float wage;

   cout << "Wage calculation" << endl;
   cout << "================" << endl;
   cout << "Starting time: ";
   cin >> start;
   cout << "Finishing time: ";
   cin >> finish;

   if (start < 6)
      if (finish <= 6)
         wage = (finish - start) * AFTERNOON_RATE;
      else
         wage = (6 - start) * AFTERNOON_RATE + (finish - 6) * EVENING_RATE;
   else
      wage = (finish - start) * EVENING_RATE;

   cout.setf(ios::fixed);
   cout.precision(2);
   cout << "The payment is R " << wage << endl;

   return 0;
}
```

Activity 12.b

Vertebrates (i.e. animals with backbones) can be divided into five main categories, namely mammals, birds, reptiles, amphibians and fish.

Write a program to classify any vertebrate into one of these five categories by asking the user the minimum number of questions.

For example, say the user wants to determine what category a snake belongs to. Without asking the user what animal he/she is thinking of, the program could ask whether the animal is warm-blooded. After the user enters 'N' for No, the program could ask whether it ever breathes through gills in its life. When the user enters 'N' for No, the program can announce that the animal is a reptile.

Of course, the questions used to split the categories have to be chosen carefully. Consider trying to distinguish birds from mammals. It is no use asking whether the animal can fly, because some birds (like penguins) can't fly and some mammals (like bats) can. Even asking whether it lays eggs is not good enough, because the Duck-billed Platypus lays eggs, and it is classified as a mammal. In fact, the best question to distinguish between a bird and a mammal is whether it suckles its young.

Test yourself

Before you give up too quickly on this program, give it a try. It's actually easier than you think. All you need to do is use a few more statements in different parts of each of the `if` statements that you use. In other words, you will need braces.

If you are unsure about using braces with nested `if` statements, or you don't know how to test for the different categories of animals, do the subactivities below.

Subactivity 12.b.i

Consider the following two `if` statements:

A.
```
if (x > 0)
   if (y > x)
      cout << "y > x > 0" << endl;
   else
      cout << "x > 0 and y <= x" << endl;
```

B.
```
if (x > 0)
{
   if (y > x)
      cout << "y > x > 0" << endl;
}
else
   cout << "x > 0 and y <= x" << endl;
```

What will be displayed by A and B respectively if `x` and `y` have the following values?

(i) -2 and 0 (ii) 2 and 0 (iii) 2 and 5

Subactivity solution

A.	(i) No output The first `if` does not have an `else` part, so no instructions will be executed if `x > 0` is `false`.	(ii) `x > 0 and y <= x` `x > 0` is `true`, so the inner `if..else` statement is executed. Here, the condition `y > x` is `false`, causing the `else` part to be executed.	(iii) `y > x > 0` The first part of the inner `if..else` statement is executed.
B.	(i) `x > 0 and y <= x` In this case, the `else` part belongs to the first `if`, so the `else` part is executed if `x > 0` is `false`.	(ii) No output The inner `if` statement is executed, and since `y > x` is `false`, no further statement is executed. The inner `if` does not have an `else` part.	(iii) `y > x > 0` The statement of the inner `if` is executed.

Discussion

The braces that are included in code B therefore make a big difference to the execution of the `if` statements. In code A, the `else` part belongs to the inner `if`, whereas in code B it belongs to the first `if`.

How do we know which `if` *statement an* `else` *belongs to?*

C++ always assumes that an `else` belongs to the nearest preceding `if`, which does not have another `else` of its own. In other words, if you want to use an `if` statement without an `else` part in the first part of an `if..else` statement, the nested `if` must be placed between braces.

Subactivity 12.b.ii

In your study notebook, write down questions to distinguish between the five kinds of vertebrates: mammals, birds, fish, amphibians and reptiles. It's easier if you "divide and conquer" by designing a question to split all vertebrates into two groups, to distinguish the mammals and birds from the fish, amphibians and reptiles. Then design questions to distinguish mammals from birds, and reptiles from fish and amphibians, and finally fish from amphibians.

Subactivity solution

These are the questions we came up with:

```
To distinguish mammals and birds from fish, amphibians and reptiles:
   Is the animal warm-blooded?

To distinguish mammals from birds:
   Does it suckle its young?

To distinguish reptiles from fish and amphibians:
   Does it ever breathe through gills in its life?

To distinguish fish from amphibians:
   Does it breathe though gills all of its life?
```

Subactivity 12.b.iii

Write a few lines of C++ code to make the first decision, i.e. to decide whether the user is thinking of a mammal or bird, or a reptile, amphibian or fish. The code must ask the user a single question, input an answer, and use an `if` statement to branch to two parts (which need not be specified).

Subactivity solution

```
cout << "Is it warm-blooded (Y/N)? ";
cin >> answer;
if (answer == 'Y')
   :
else
   :
```

Easy, hey?

Now complete the main activity.

Activity solution

```
//Classifies any animal into one of five groups
#include <iostream>
using namespace std;

int main( )
{
   char answer;

   cout << "This program will classify any vertebrate that you can think of." << endl;
   cout << "Think of a vertebrate animal" << endl;
   cout << "Is it warm-blooded (Y/N)? ";
   cin >> answer;
   if (answer == 'Y')
   {
      cout << "Does it suckle its young (Y/N)? ";
      cin >> answer;
      if (answer == 'Y')
         cout << "It's a mammal" << endl;
      else
         cout << "It's a bird" << endl;
   }
   else
   {
      cout << "Does it ever breathe through gills during its life (Y/N)? ";
      cin >> answer;
      if (answer == 'Y')
      {
         cout << "Does it breathe through gills all of its life (Y/N)? ";
         cin >> answer;
         if (answer == 'Y')
            cout << "It's a fish" << endl;
         else
            cout << "It's an amphibian" << endl;
      }
      else
         cout << "It's a reptile" << endl;
   }
   return 0;
}
```

Discussion

Note the following about the above code: We could have written the program to ask all the questions at the beginning (storing the answers in separate variables) and then used `if` statements to decide what

category of animal is represented by the answers given. However, the program would then be asking more questions than it should (because some of them would be totally irrelevant). To keep the number of questions to a minimum, some questions are placed within nested `if` statements.

In fact, this program illustrates an important principle of programming: Although it is generally useful to identify the input, processing and output of any program that you want to write (as we did in previous lessons and in the solution to the first main activity in this lesson), it is not always best to keep these in watertight compartments in the code of the program. In this program we do not have separate code for input, processing and output. In fact, it would be impossible to do so. Input, processing and output are all combined with one another, mainly because further input and output depends on previous processing.

Activity 12.c

The South African Post Office publishes a pamphlet every now and again with revised postal rates. Here is an extract from the 2011 brochure with the internal postal rates for letters (sent by ordinary mail):

Size	Cost
235 x 120 x 5mm	R2.50
250 x 176 x 10mm	R5.00
353 x 250 x 30mm	R6.25

If the dimensions of a letter exceeds any of the restrictions of a particular category, it falls into the next category. If it does not comply with the restrictions of any category, it is regarded as a parcel and its postal rate depends on its mass.

Write a program for your local Post Office to determine the postal rate for a letter. The program should read in the dimensions of a letter (length, width and thickness) and then display the postal rate (or a message to say that the letter must be regarded as a parcel).

Test yourself

This program is quite a bit easier than the preceding two activities. Before you attempt the subactivities below, make an attempt to write the program. Then compare your answer with the solution we give. If you answer looks different (especially the indentation) then work through the subactivities below.

Subactivity 12.c.i

Change the indentation of the solution to Subactivity 12.a.i so that the `else` statements are directly under one another.

Subactivity solution

The `if` statement will look as follows:

```
if (x > 0)
   numPos = numPos + 1;
else if (x < 0)
   numNeg = numNeg + 1;
else
   numZero = numZero + 1;
```

Discussion

When successive nested `if` statements are in the `else` parts of previous `if` statements, it is quite acceptable to place the `else` statements directly under one another, with the next `if` right next to the `else`. This makes the whole structure more readable.

Subactivity 12.c.ii

Say you want to award symbols for examination marks as follows:

90 and higher	A
80 to 89	B
70 to 79	C
60 to 69	D
less than 60	E

Why will the following `if` statement not perform the task successfully? Rewrite the `if` statement so that it will work correctly.

```
if (mark >= 60)
   symbol = 'D';
else if (mark >= 70)
   symbol = 'C';
else if (mark >= 80)
   symbol = 'B';
else if (mark >= 90)
   symbol = 'A';
else
   symbol = 'E';
```

Subactivity solution

The symbol `'D'` will be awarded for all marks from 60 upward. The first condition will be `true`, and so the statement in the first part will be executed. The rest of the alternatives will then be ignored. To solve the problem, the most restrictive condition must be tested first. The following code will work properly:

```
if (mark >= 90)
   symbol = 'A';
else if (mark >= 80)
```

```
      symbol = 'B';
   else if (mark >= 70)
      symbol = 'C';
   else if (mark >= 60)
      symbol = 'D';
   else
      symbol = 'E';
```

Note once again that we have placed each successive `else` part directly under the previous one, rather than indent each one a further number of spaces. Nested `if` statements in this special form are called *multiple alternative decisions*.

Now try to do the main activity. You should use multiple alternative decisions for it.

Activity solution

One possible solution is:

```
//Determine postal tariff given dimensions of a letter
#include <iostream>
using namespace std;

int main( )
{
   int length, width, thickness;
   float rate;

   cout << "Enter the length of the envelope: ";
   cin >> length;
   cout << "Enter the width of the envelope: ";
   cin >> width;
   cout << "Enter the thickness of the envelope: ";
   cin >> thickness;

   if (length <= 235 && width <= 120 && thickness <= 5)
      rate = 2.50;
   else if (length <= 250 && width <= 176 && thickness <= 10)
      rate = 5.00;
   else if (length <= 353 && width <= 250 && thickness <= 30)
      rate = 6.25;
   else
      rate = -1;

   if (rate != -1)
      cout << "The postal rate is R" << rate << endl;
   else
      cout << "Too large - regard as parcel" << endl;

   return 0;
}
```

Note: We could have got by without variable `rate` in which case we would have needed four `cout` statements, one for each rate and one to handle the parcel problem.

We could have (and perhaps should have) declared a number of constants instead of using literal values in the above program. For example, the Post Office changes there tarriffs every year, and it would be good to be able to change the constants at the beginning of the program, rather than have to search through the code to make such changes.

Important points in this lesson

Programming concepts

In this lesson we looked again at `if` and `if..else` statements, and particularly at the use of nested `if` statements.

Simple `if` statements

In Lesson 8 you encountered simple `if` statements with the following structure:

The `if` statement:

```
if (Condition)
   Statement1;
```

The `if..else` statement:

```
if (Condition)
   Statement1;
else
   Statement2;
```

Condition is tested, and if it is `true`, *Statement1* is executed. If it is `false` and there is an `else` part, *Statement2* is executed, otherwise the statement following the `if` statement in the program is executed. *Statement1* and *Statement2* can also be compound statements (i.e. a number of statements placed between braces). These statements may also be other `if` statements, in which case we are dealing with nested `if` statements.

Nested `if` statements

A	B	C

`if (Condition1)`
`  if (Condition2)`
`  Statement1;`
`  else`
`  Statement2;`
`else`
`  Statement3;`
	`if (Condition1)`
	`  if (Condition2)`
	`  Statement1;`
	`  else`
	`  Statement2;`
		`if (Condition1)`
		`{`
		`  if (Condition2)`
		`  Statement1;`
		`}`
		`else`
		`  Statement2;`

The first part of the first `if..else` statement in A contains another `if..else` statement. Since there are two `if` statements and two `else` parts, it is fairly clear which `else` part belongs to which `if` statement. On the other hand, statement B is an `if` statement with a nested `if..else` statement in it. It is clear from

the way we laid it out that the `else` belongs to the inner `if` statement, but how does the C++ compiler determine this? The rule is that an `else` belongs to the nearest `if` preceding it, which does not have another `else` part of its own. If we want the `else` part in code B to belong to the first `if`, we must place the nested `if` statement between braces. Code C shows what the resulting code should look like.

Multiple alternative decisions

Nested `if` statements can become quite complex when you have to make provision for more than three alternative options. In the situation where all the `else` parts (except perhaps the last one) are followed by an `if..else` statement (as in the solution to Subactivity 12.a.ii), the nested `if` can be coded as a *multiple alternative decision*, with the following structure:

```
if (Condition1)
   Statement1;
else if (Condition2)
   Statement2;
else if (Condition3)
   Statement3;
:
else if (ConditionM)
   StatementM;
else
   StatementN;
```

An `if` statement of this form is interpreted as follows: The conditions *Condition1*, *Condition2*, ... are evaluated from the top until one of the conditions is `true`. The statement in the corresponding `if` statement is then executed and the rest of the multiple alternatives are ignored. If all the conditions are `false`, the statement in the `else` part of the last `if` statement is executed.

Simplification of `if` statements

Although we haven't seen any examples of this in this lesson, think about this:

Say an `if` statement has the following structure:

```
if (Condition1)
   if (Condition2)
      if (Condition3)
         Statement;
```

Such an `if` statement can often be simplified by using the boolean operator `&&`:

```
if (Condition1 && Condition2 && Condition3)
   Statement;
```

Programming principles

Although the way in which C++ interprets nested `if` statements is not influenced by their layout, the indentation is important for the readability of the code. The same principle as explained in Lesson 8 applies: Each `else` should be placed directly under the `if` that it belongs to, and all the statements within them should be indented the same distance.

One exception to this rule is the indentation of a multiple alternative decision. Each successive `else` part is not indented a further number of spaces, as we would normally do. It is quite acceptable to place each

of the successive `else if ...` statements directly under one another as it makes the whole structure more readable.

There is no restriction on the depth to which `if` statements may be nested, but the depth of nesting should not make the code difficult to read.

Nested `if` statements are often more efficient than sequential `if`s.

Exercises

Exercise 12.1

Write a program that reads in three numbers and determines whether the sum of any two of the numbers is equal to the remaining number.

Exercise 12.2

Write a program that reads in two numbers representing the throw of a pair of dice. If a total of 7 or 11 is entered, display the message "You win!". If the total is 2, display "Snake eyes!", if the total is 12, display "Good shot!", otherwise display "Try again."

Exercise 12.3

Write a program that determines whether a year is a leap year. A leap year is a year which is divisible by 4, and if it is divisible by 100, it must also be divisible by 400.

Exercise 12.4

A big supermarket has the following special offer on all breakfast cereals: Depending on the amount of money spent by a customer, a certain discount is awarded. The discount is determined as follows:

Amount spent	Discount
less than R50	10%
R50 to just under R70	20%
R70 to just under R100	30%
R100 to just under R200	40%
R200 and over	50%

Write a program that inputs the amount spent and determines and displays the discounted price.

Lesson 13

Switch statements

Purpose of this lesson

In the last activity of the previous lesson we encountered the multiple alternative decision form of an `if` statement, in which precisely one of a number of alternative options is chosen. The option is chosen according to the evaluation of one or more boolean expressions. In this lesson, we consider another means of handling similar problems. In certain situations, a `switch` statement is more suitable to solve the problem.

Activity 13.a

Write a program that inputs two floating point numbers and then performs one of four operations on them: addition, subtraction, multiplication or division. The program must ask the user which one of the operations to perform (specified by entering a single character: `A`, `S`, `M` or `D`).

For example, if the user enters the values 10.0 and 50.0 and asks that multiplication be performed on them, the program should display the value 500.0 as output.

Remember that division by 0 is not allowed, and the program will therefore have to test for this in the division option.

Use a `switch` statement to perform the correct operation and display the result.

> **Test yourself**
>
> If you are not sure how to use a `switch` statement for this problem, work though the subactivities below before you tackle the activity.

Subactivity 13.a.i

Consider the following two examples of `switch` statements:

Example 1:

```
switch (day)
{
   case 1:
      cout << "Sunday" << endl;
      break;
   case 2:
      cout << "Monday" << endl;
```

```
      break;
   case 3:
      cout << "Tuesday" << endl;
      break;
   case 4:
      cout << "Wednesday" << endl;
      break;
   case 5:
      cout << "Thursday" << endl;
      break;
   case 6:
      cout << "Friday" << endl;
      break;
   case 7:
      cout << "Saturday" << endl;
      break;
   default:
      cout << "Invalid day" << endl;
}
```

Example 2:

```
switch (day)
{
   case 1:
   case 7:
      cout << "It's weekend!" << endl;
      if (day == 1)
         cout << "Have a picnic!" << endl;
      else
         cout << "Go shopping!" << endl;
      break;
   case 2:
   case 3:
   case 4:
   case 5:
   case 6:
      cout << "Weekday" << endl;
      break;
   default:
      cout << "Invalid day" << endl;
}
```

For each of these two code fragments, predict what will be displayed if

(i) `day` has the value 1

(ii) `day` has the value 4

(iii) `day` has the value 10

Subactivity solution

Value of `day`	Output for Example 1	Output for Example 2
(i) 1	`Sunday`	`It's weekend!` `Have a picnic!`
(ii) 4	`Wednesday`	`Weekday`
(iii) 10	`Invalid day`	`Invalid day`

And you thought you didn't know anything about `switch` statements!

Discussion

You have probably been wondering what the purpose and meaning is of the `break` statements that are used with `switch`.

The `break` statement can be used to break out of any control structure (i.e. not just out of a `switch`) prematurely. Consider the following `while` loop:

```
cin >> number;
check = number - 1;
while (check > 1)
{
   if (number % check == 0)
      break;
   check--;
}
if (check == 1)
   cout << number << " is a prime number!" << endl;
```

This loop will not only terminate when `check` gets to 1. It might terminate earlier (if `number` is a multiple of `check`) since the `break` command (if executed) will cause the loop to stop repeating prematurely.

(By the way, the above `while` loop can also be written as

```
cin >> number;
check = number - 1;
while (check > 1 && number % check != 0)
   check--;
if (check == 1)
   cout << number << " is a prime number!" << endl;
```

We prefer this to the version using `break` because we can see exactly under what conditions the `while` loop will terminate from its condition.)

But this doesn't explain why we need the `break` command with `switch`. The thing is that as soon as one case matches the selector *all the subsequent cases are also executed.* The `switch` statement is designed this way to allow the same code to be executed for multiple cases, as in Example 2 above.

The `default` section is always executed (unless of course a `break` has been encountered earlier). It is used for code that must be executed if the selector does not match any of the cases.

Subactivity 13.a.ii

In your study notebook, write a `switch` statement that uses a variable `age` and displays a message indicating what type of vehicle a person of that age may drive. The `switch` statement must code the following categories:

Age	Vehicle
1 to 4	Tricycle
5 to 15	Bicycle
16 to 17	Motorcycle
18 and older	Motor car

Subactivity solution

```
switch (age)
{
   case 1:
   case 2:
   case 3:
   case 4:
      cout << "You can ride a tricycle" << endl;
      break;
   case 5:
   :
   case 15:
      cout << "You can ride a bike" << endl;
      break;
   case 16:
   case 17:
      cout << "You can ride a motorcycle" << endl;
      break;
   default:
      cout << "You can drive a car" << endl;
}
```

Discussion

Say the `switch` statement above is used in a program where the age is more often than not 16 or 17. In this case it would be better to place these options first and thus prevent the selector from being tested against all the options before the most common alternative is reached.

In other words, if it is clear from the context of the problem that certain options of a `switch` statement will be chosen more often than others, they should be placed at the beginning of the `switch` statement to improve efficiency of the program.

You should now be able to tackle the main activity. Note that you will need an `if` statement inside one of the `case` sections (i.e. where division is done, to prevent division by 0) as in Example 2 above.

Activity solution

One possible solution to the problem is:

```
//Simulates a simple calculator
#include <iostream>
using namespace std;

int main( )
{
   float value1, value2, answer;
   char operation;
   bool error;

   //Input two values and the type of operation
   cout << "Enter the first value: ";
   cin >> value1;
   cout << "Enter the second value: ";
   cin >> value2;
   cout << endl << "Which operation should be performed?" << endl;
   cout << "(A)ddition" << endl;
   cout << "(S)ubtraction" << endl;
   cout << "(M)ultiplication" << endl;
   cout << "(D)ivision" << endl;
   cout << "Type the first letter of the one you choose: ";
   cin >> operation;

   //Do appropriate calculation
   error = false;
   switch (operation)
   {
      case 'a':
      case 'A':
         answer = value1 + value2;
         break;
      case 's':
      case 'S':
         answer = value1   value2;
         break;
      case 'm':
      case 'M':
```

```
         answer = value1 * value2;
         break;
      case 'd':
      case 'D':
         if (value2 == 0)
            error = true;
         else
            answer = value1 / value2;
         break;
      default:
         error = true;
   }

   //Output result
   if (!error)
      cout << "The answer is " << answer << endl;
   else
      cout << "Illegal operation" << endl;

   return 0;
}
```

Important points in this lesson

Programming concepts

In this lesson we looked at the `switch` statement. A `switch` statement is used when a program must choose between more than two possible routes, and precisely one of the alternative options must be chosen.

The structure of a `switch` statement is as follows:

```
switch (Selector)
{
   case Label1:
      Statements1;
   case Label2:
      Statements2;
   :
   case LabelN:
      StatementsN;
   default:
      StatementsD;
}
```

The *Selector* expression must be of an ordinal data type. In other words, it may not be a floating point expression or a character string.

Label1, *Label2*, ... and *LabelN* may only take values of the same type as *Selector*.

The same value may not occur in more than one of the labels.

Statements1, *Statements2*, ... can be zero or more C++ statements, including `if` statements or `while` loops. It is not necessary to place multiple statements in braces.

The value of *Selector* is compared to each of the labels (from the top) until the value of *Selector* matches a *Label*. If the value of *Selector* matches *Label3* for example, *Statements3* will be executed.

All statements in *all* cases under a matching label are executed. The `break` command must therefore be used to break out of the `switch` statement if you want to prevent further statements in subsequent cases from being executed.

The `default` part is optional. If present, it is always executed unless a `break` has been encountered beforehand.

A `switch` statement can only be used if all the alternatives depend on the value of the same ordinal variable (or expression). If the boolean expressions on which the choices are made depend on different variables or on a floating point variable, nested `if` statements must be used. Any `switch` statement can be replaced by a (nested) `if` statement, but not always the other way round.

Programming principles

We generally place the options which are chosen more often than the others at the beginning of the `switch` statement to avoid unnecessary testing.

When the use of nested `if` statements reduces the readability of code, and the problem is suitable for using a `switch` statement, a `switch` statement should rather be used. Deep nesting makes code difficult to understand, so avoid it where possible.

Exercises

Exercise 13.1

Your child wants to go to university and you must consider all the options. You state the case as follows to her: If she obtains 90% of higher, she can go to any university of her choice and you will give her a car. If she obtains from 75% to 89%, and she earns more than R5 000 during the December holidays, she can also go the university of her choice and you will give her a car. If she obtains more than 74% but does not earn enough money, she can study at the university of her choice, but you will not give her a car. If she obtains less than 75%, but more than 59%, she must study at the nearest university. With less than 60% she cannot go to university and will have to consider other alternatives.

Write a program to read in your daughter's average mark, and the amount that she earned in her holiday job, and display a message indicating which university she may attend.

The program must use a `switch` statement.

Exercise 13.2

Do Exercise 12.2 again and make use of a `switch` statement in the program. The question was as follows:

> Write a program that reads in two numbers representing the throw of a pair of dice. If a total of 7 or 11 is entered, display the message "You win!". If the total is 2, display "Snake eyes!", if the total is 12, display "Good shot!", otherwise display "Try again."

Exercise 13.3

Write a program to help the cashier at a parkade determine the amount of money owed by a user of the parkade. The program must ask whether the vehicle is a motorcar or a truck and then read in the number of hours that the person was parked. Any part of an hour is considered as a whole hour. The charges are: R2 for the first hour, R3 for 2 hours, R5 for 3 to 5 hours and R10 for more than 5 hours. An extra R1 is

added for trucks. Assume that the vehicle will not be parked for longer than 24 hours in the parkade.

Exercise 13.4

Write a `switch` statement that uses two integer variables `month` (1 to 12) and `year` and displays the number of days in that month. Leap years must be taken into account. (Hint: Exercise 12.3 explains how to determine whether a year is a leap year. For February you will therefore need to use nested `if` statements inside the `switch` statement.)

Lesson 14

More while loops

Purpose of this lesson

In Lesson 9 we introduced one iteration structure, namely the `while` loop. Since you should already be able to use `while` loops, the purpose of this lesson is primarily to refresh your memory.

Instead of dividing the lesson into main and subactivities, we give a sequence of activities each of which addresses an important aspect of the `while` loop.

Only the last activity, which covers a variation of the `while` loop called the `do..while` loop, has a number of subactivities.

Activity 14.a

Read through Lesson 9 that deals with the `while` loop again.

Now, write a `while` loop that displays each of the integers from 1 to 5, together with its square, on a separate line. Assume that variables `x` and `xSq` are declared as integers.

Activity solution

The required `while` loop looks as follows:

```
x = 1;
while (x <= 5)
{
   xSq = x * x;
   cout << x << " squared is " << xSq << endl;
   x++;
}
```

Discussion

In this loop, `x <= 5` is the *loop condition*, and `x` is the *loop control variable*. Note how the three rules for `while` loops have been applied here:

1. `x` is initialised to 1 before the loop is encountered. If we don't do this, `x` will contain an unknown integer value. If this unknown value incidently exceeds 5, the statements inside the loop will never be executed. If it is 0, the statements will be executed once too many. The question requires that the values from 1 to 5 are used, therefore we initialise `x` to 1.

2. The value of `x` is tested in the condition of the loop. The loop will continue executing while `x` is less than or equal to 5.
3. Inside the loop the value of x is changed by adding 1 to it. If we leave out the statement `x++;` (or `x += 1;` or `x = x+1;`) the loop will keep on executing forever, since `x`'s value will remain 1 and the loop condition will never become `false`.

So when using a `while` loop, remember to always apply the three rules:

1. The variable(s) that appear in the loop condition (i.e. the loop control variable(s)) must be initialised when the `while` loop is first encountered.
2. Test the loop control variable(s) in the condition of the loop. The condition must specify the values of the control variable(s) for which the loop must continue repeating, and hence (implicitly) the values for which the loop must terminate.
3. Inside the body of the loop, the value of the loop control variable(s) should be changed to ensure that the loop condition becomes `false` at some stage.

This is sometimes called the ITC principle (for *Initialise-Test-Change*).

Activity 14.b

Consider the `while` loop below. How many times will the loop repeat? What is displayed on each iteration? What is displayed after the loop has ended?

```
int x = 3;
int count = 0;
while (count < 3)
{
   x = x * x;
   cout << x << endl;
   count++;
}
cout << count << endl;
```

Activity solution

The loop is executed 3 times - when the value of `count` is respectively equal to 0, 1 and 2. When `count` equals 3, the loop condition is `false` and the loop is exited.

The output that will be produced by the loop is:

```
9
81
6561
```

Initially `x` has the value 3. The first time the statement `x = x * x;` is executed, `x` is assigned the value 9. With the second iteration of the loop body, `x` is assigned the value 9 * 9 = 81, and with the third iteration it gets the value 81 * 81 = 6561.

When the loop condition is `false`, the `cout` statement that follows the `while` statement is executed. It displays the value 3, which is the value of `count` when the loop ends.

Discussion

A loop such as the one above is called a *counter-driven loop*, since its iteration is determined by a variable that keeps count of the number of times the loop is executed. When this variable reaches a certain value, the loop ends. To use a counter-driven loop the programmer should know in advance the exact number of times the loop must be repeated. The counter variable (such as `count` above) is manipulated by the loop in such a way that the loop is executed the required number of times.

Activity 14.c

Write a `while` loop that inputs the monthly rainfall figures for a year, and calculates and displays the total rainfall (in millimetres) for the year.

Activity solution

We know that we must input twelve values (one for each month), and therefore use a counter-driven loop with `month` as counter variable. Variable `month` is initialised to 1, and inside the loop it is repeatedly increased by 1. When `month`'s value is 13, execution of the loop ends.

```
float totalRain = 0.0;
int month = 1;
while (month <= 12)
{
   cout << "Enter rainfall for month " << month << ": ";
   cin >> rainfall;
   totalRain += rainfall;
   month++;
}
cout << "The total rainfall for the year is " << totalRain << endl;
```

Discussion

When a loop is used to accumulate the sum of a sequence of values, there is a variable that acts as an *accumulator*. The values of which the sum is to be calculated, are added to the accumulator variable one-by-one with each iteration of the loop. In this code `totalRain` acts as the accumulator. It is important that the accumulator is initialised to 0 before the loop. If the statement `totalRain += rainfall;` is executed during the first iteration of the loop, `totalRain`'s value must be 0 to ensure that an unwanted value is not initially added to the accumulated sum.

Note how variable `month` acts as a counter, to ensure that the program inputs exactly 12 values.

Not all `while` loops are counter-driven. We generally use a `while` loop when we cannot determine in advance the number of times the loop will execute. The continued repetition of such a loop will depend on whether a certain condition is `true` or `false`. In the activity below we use such a loop.

Activity 14.d

The number of people that live in a city increases by 10% every year. Write a program that inputs a value for the current population, and then uses a `while` loop to determine how many years it will take before the population exceeds 1 000 000.

Activity solution

One possible solution is

```
//Calculates how many years it will take for the population
//to exceed 1 million
#include <iostream>
using namespace std;

int main( )
{
   float population;
   int yearCount = 0;
   const float LIMIT = 1000000;

   cout << "How many people currently live in the town? ";
   cin >> population;

   while (population <= LIMIT)
   {
      yearCount++;
      population = population * 1.1;
   }

   cout << "In " << yearCount << " years";
   cout << " the population will exceed " << LIMIT << endl;

   return 0;
}
```

Note that we declared `population` as a floating point number even though the number of people in a city should always be an integer. The reason is that we want to multiply by 1.1. If we make population an `int`, the compiler will complain about the statement `population = population * 1.1;` because `population * 1.1` will give a floating point value (even though `population` is an `int`). The compiler will complain about the attempt to assign a floating point value to an integer variable because information will be lost.

Discussion

Although this loop contains the variable `yearCount` which acts as a counter, it is not a counter-driven loop. The number of iterations does not depend on this counter. (We use `yearCount` to determine the number of times the loop executes by adding 1 to it with each iteration. It is therefore necessary to initialise `yearCount` to 0 before the loop.)

The loop control variable of this loop is `population`. Inside the loop `population` is increased by 10%. As long as `population`'s value is less than or equal to 1 000 000, the loop repeats. Because `population`'s value increases with each iteration, we know that sooner or later it will reach a value greater than 1 000 000.

A loop such as the `while` loop above is sometimes called an *situation-driven loop*. The loop ends when a specific situation is reached. In this case, the situation is when the population exceeds 1 million. The number of iterations of the loop thus depends on the value that the program initially inputs into `population`.

The three rules for `while` loops also apply to situation-driven loops:

1. The loop control variable(s) must be initialised before the loop.
2. The loop control variable(s) must be tested in the condition of the loop.
3. The values of the loop control variable(s) must be changed inside the loop.

Note that it is possible that the statements inside the body of a `while` loop are never executed. Say, for example, that the value 2 000 000 is input for `population` when the program above executes. The loop condition `population <= LIMIT` is immediately `false` and the program jumps directly to the `cout` statement after the loop.

Activity 14.e

Write a program that inputs a list of values for variable `x`, and for each of these values it calculates the value of `y` using the formula: $y = x^3 - 3x + 1$. For example, if `x` is 10, then `y` should get the value 10^3 - 3(10) + 1 = 1000 - 30 + 1 = 971.

For each value of `x`, the program must display `x` and the calculated value of `y` on one line. The input list ends when the value 0 is entered for `x`.

Activity solution

Here is one possible solution:

```
//Calculates the value of a polynomial
#include <iostream>
using namespace std;

int main( )
{
   float x, y;

   //Inputs the first value for x
   cout << "Enter a value for x: ";
   cin >> x;

   while (x != 0)
   {
```

```
      //Calculate y, and display x and y
      y = (x*x*x) - (3*x) + 1;
      cout << "x = " << x << " y = " << y << endl << endl;

      //Input the next value for x
      cout << "Enter the next value for x: ";
      cin >> x;
   }

   return 0;
}
```

Discussion

We often use loops like this to input a sequence of values. If it is known in advance exactly how many values appear in the list, we can use a counter-driven loop. The exact number of values might, however, be unknown. One way to handle such a situation is to ask the user to enter a unique value after the last value in the input list has been given. This unique value is called the *sentinel*. In the above loop, 0 acts as the sentinel to indicate that no further values need to be processed.

A `while` loop that executes until a sentinel value is input is called a *sentinel-driven loop*.

Activity 14.f

Lynette wants to adapt the following program so that it displays the final value of variable `value` after the `while` loop:

```
//Inputs integers until their sum reaches a given target
#include <iostream>
using namespace std;

int main( )
{
   int sum = 0;
   int target;

   cout << "Enter the target: ";
   cin >> target;

   while (sum < target)
   {
      int value;
      cout << "Enter a value: ";
      cin >> value;
      sum += value;
   }

   if (sum == target)
      cout << "Target reached!" << endl;
   else
```

```
      cout << "Target exceeded!" << endl;

   return 0;
}
```

However, if she adds the statement `cout << "The last value was " << value << endl;` after the loop, the program won't compile any more. An error message is displayed, namely

```
value undeclared
```

Lynette then declares `value` as an integer at the beginning of the program, but then the output is

```
The last value was 4198571
```

even though that definitely wasn't the last value she entered. Explain what the problem is and help her to fix it.

Activity solution

The compiler error occurs because variable `value` is declared inside the body of the loop, making it inaccessible outside the loop. The problem with declaring `value` outside (before) the loop is that we now have two instances of variable `value`. Normally, the compiler won't allow us to declare two variables with the same name, but in this case, since one is declared inside the loop and the other outside, the compiler can distinguish which variable is being referred to. So any references to `value` in the body of the loop are matched to the variable declared inside the loop, whereas any references to `value` outside the loop are matched to the variable declared outside (i.e. before) the loop.

Since variable `value` declared outside the loop is never initialised, when it is referenced outside (i.e. after) the loop, its value is garbage.

The solution to Lynette's problem is to declare `value` outside the loop and remove the declaration inside the loop.

Discussion

The original variable `value` is called a *block variable* because it is declared in the body of the `while` loop. A *block* is another name for a statement sequence, i.e. a series of statements enclosed by braces { and }. A block variable is only accessible within the block. The compiler will complain about any attempt to access a block variable outside its block.

If there is another variable with the same name in the function in which the block resides, the block variable *hides* the outer variable for the duration of the block. In other words, any references to the variable name in the block are deemed to be to the block variable and the outer variable is hidden. This can cause no end of confusion, and so our general rule is: Don't declare variables inside blocks with the same names as variables declared outside them.

In fact, block variables are not a very good idea for another reason: When a program reaches a variable declaration, memory is made available for the variable. When the program reaches the end of the block in which the variable is declared, the variable is destroyed and it becomes inaccessible.

Making memory available for variables and destroying them requires quite a bit of work by the computer (behind the scenes - we don't see what's going on). This is why a block variable in a loop is generally not a good idea - it can make a program inefficient because the variable has to be created and destroyed each time the body of the loop is repeated.

Activity 14.g

Type in the following program and get it to run. See if you can get it to terminate normally, i.e. to display the final message. Add a prompt message to the program to help the user to use the program correctly.

```
// Calculate the average of a list of integers
#include <iostream>
using namespace std;

int main( )
{
   int value, sum, many;
   float average;
   sum = 0;
   many = 0;

   while (cin >> value)
   {
      sum += value;
      many++;
      cout << "The sum so far is " << sum << endl;
   }

   average = float(sum) / many;
   cout << "The average is " << average << endl;

   return 0;
}
```

Activity solution

This is a somewhat unfair question. If you couldn't work it out, look at the program below and try again:

```
// Calculate the average of a list of integers
#include <iostream>
using namespace std;

int main( )
{
   int value, sum, many;
   float average;
```

```
   sum = 0;
   many = 0;

   cout << "Enter a list of numbers (<Ctrl+D> to end)" << endl;
   while (cin >> value)
   {
      sum += value;
      many++;
      cout << "The sum so far is " << sum << endl;
   }

   average = float(sum) / many;
   cout << "The average is " << average << endl;

   return 0;
}
```

The trick is to press <Ctrl+D> to indicate the end of input.

This use of a `while` loop is quite strange for a number of reasons. It is strange because it doesn't appear to have a control variable. You might think that `value` is the control variable, but it isn't because it's not the value of `value` that determines when the loop should terminate. (You can check this by outputting the final value of `value` after the loop has terminated.) Also, the Initialise-Test-Change (ITC) principle does not seem to be applied: A control variable is not initialised before the loop, it is not tested in the condition of the loop, and its value is not changed in the body of the loop.

What is even more strange is that a statement that normally stands on its own (i.e. `cin >> value`) is used as the condition of the `while` loop.

The control variable of this `while` loop is actually the `cin` object, which is declared in C++'s standard header file `iostream`. Every time `cin` successfully inputs a value, it returns `true`. As soon as `cin` tries to input a value and fails (when the user presses <Ctrl+D>), its value becomes `false`. So we see that `cin` is initialised, tested and changed, all in the condition of the loop.

Although we don't like this trick very much, it is a technique that you may well come across in C++ programs that other people write. We don't like it because it is not very user-friendly to require the user to press <Ctrl+D>, and it doesn't comply with our ITC rule for `while` loops very nicely. Its advantage is that it makes code quite a bit shorter.

Activity 14.h

The number of digits in an integer can be determined by counting the number of times the integer can be divided by 10 before the quotient is 0. Write a program to count the number of digits in a given integer. Your program should use a `do..while` loop.

Test yourself

You probably haven't seen a `do..while` loop before. It's just like a `while` loop except that the condition is placed at the end of the loop instead of the beginning.

The first subactivity below gives an example of a `do..while` loop, and explains the major difference with a normal `while` loop.

Subactivity 14.h.i

Consider the `while` loop and the `do..while` loop below:

```
cin >> c;
while (c <= 10)
{
   cout << "Line " << c << endl;
   c++;
}
```

```
cin >> c;
do
{
   cout << "Line " << c << endl;
    c++;
} while (c <= 10)
```

Will these two loops always give the same output?

Subactivity solution

The answer is No - the loops will not give the same output for all values of `c`. If `c`'s value is initially greater then 10, the `while` loop will not execute at all. The `do..while` loop, however, will produce one line of output. The reason is that the `while` loop's condition is tested before the loop executes, whereas the condition of the `do..while` loop is tested after each iteration of the loop. We therefore call the `while` loop a *pretest loop*, and the `do..while` loop a *posttest loop*. A `do..while` loop is thus only suitable if the statements included in it must be executed at least once.

Discussion

You will recall that there are three rules for writing a `while` loop – the ITC priciple. These three rules also apply to `do..while` loops, except that sometimes you can get away without rule 1, namely to initialise the loop control variable(s). Since the condition is at the end the loop, the body of the loop can often be used to give the loop control variable a value.

Subactivity 14.h.ii

The `while` loop below determines the biggest of a number of positive integers. The input list ends with 0 or any negative number. Rewrite the code so that it uses a `do..while` loop.

```
cout << "Enter a positive integer: ";
cin >> value;
max = value;
while (value > 0)
{
   cout << "Enter a positive integer: ";
   cin >> value;
   if (value > max)
      max = value;
}
```

Subactivity solution

If we use a `do..while` loop, the code looks as follows:

```
max = 0;
do
{
   cout << "Enter a positive integer: ";
   cin >> value;
   if (value > max)
      max = value;
} while (value > 0);
```

Note how `max` is initially assigned the value 0, which will be less than any positive input values. This ensures that `max` gets the value of the first integer the user enters during the first iteration of the loop.

The following code is not equivalent to the code given above, however:

```
cout << "Enter a positive integer: ";
cin >> value;
max = value;
do
{
   cout << "Enter a positive integer: ";
   cin >> value;
   if (value > max)
      max = value;
} while (value > 0);
```

We cannot, as we did with the `while` loop, input the first value before the `do..while` loop is entered, since this will cause a problem if the user starts with 0 or a negative number. If this is the case, the value will be input outside the loop and even though it is meant to indicate the end of the input list, the statements inside the loop will still be executed (another value will be input). If the user then enters positive values, the loop will continue executing (even though the user started with 0 or a negative number). Check this yourself!

Discussion

The first solution above is a good application of a `do..while` loop since it simplifies and shortens the code. It saves having to input a value once before the body of the loop.

Subactivity 14.h.iii

Consider the following `do..while` loop:

```
do
{
   cout << "Do you want to continue? (Y/N) ";
   cin >> answer;
```

```
      if (answer != 'Y' && answer != 'N')
         cout << "Type Y or N" << endl;
   } while (answer != 'Y' && answer != 'N');
```

This code forces the user to type either `Y` or `N` (upper case only) because the `do..while` statement will keep executing until one of these values are entered. By blocking incorrect data in this way the programmer can ensure that the program will execute correctly.

Rewrite this code using a normal `while` loop.

Subactivity solution

```
cout << "Do you want to continue? (Y/N) ";
cin >> answer;
while (answer != 'Y' && answer != 'N')
{
   cout << "Type Y or N" << endl;
   cin >> answer;
}
```

Although we have to duplicate one of the statements in the body of the loop before the `while` loop, we have managed to dispense with the `if` statement that was necessary with the `do..while` loop. This happens very often when you need to validate input data. In such a situation, it doesn't really matter whether you use a `while` or a `do..while` loop. A `while` loop is perhaps slightly better because it is a bit simpler. (It doesn't require an `if` statement inside a loop construct.)

Try to do the main activity now. Remember to use a `do..while` loop.

Activity solution

```
//Calculates the number of digits in a number
#include <iostream>
using namespace std;

int main( )
{
   int number, num, count;

   //Prompt the user for a number
   cout << "Enter an integer: ";
   cin >> number;
   num = number;

   //Determine the number of digits in the number

   count = 0;
   do
   {
      count++;
```

```
      num /= 10;
   } while (num != 0);

   cout << number << " contains " << count << " digit(s)" << endl;

   return 0;
}
```

A `do..while` loop is better than a normal `while` loop in this situation because the body of the loop must always be executed at least once. Even if the user enters 0, this number consists of a single digit, so the body of the loop must be executed once.

Important points in this lesson

Programming concepts

Block variables are variables that are declared in a block. The body of a loop (or in fact any sequence of statements enclosed in braces) represents a *block*. Block variables hide variables with the same name that are declared in the enclosing function.

A shorthand way of inputting values in a loop is to use a `while` loop of the following form:

```
while (cin >> Variable)
   Statement;
```

`Variable` can be a variable of any type, and *`Statement`* can be any statement or block of statements enclosed in braces. The user has to press <Ctrl+D> to terminate input and cause the condition of the loop to become `false`.

A `do..while` loop has the following structure:

```
do
   Statement;
while (Condition);
```

`Statement` can be a single statement or a block of statements enclosed in braces.

After execution of *`Statement`*, the *`Condition`* is tested. If it is `true`, the loop is repeated, otherwise it ends.

A `do..while` loop is a *posttest loop*, which means the statements inside the loop execute before the loop condition is tested. A `while` loop, on the other hand, is a *pretest loop* - the loop condition is tested before the statements inside the loop execute.

Programming principles

Rules for using `while` and `do..while` loops

The condition of a `while` loop is always expressed in terms of one or more variables that control the number of iterations of the loop. These variable(s) are called the *loop control variable(s)*. When using a `while` loop, always make sure that you apply the following three rules with respect to these variables:

1. The loop control variable(s) must be initialised before the `while` loop is first encountered.
2. The condition of the `while` loop must test the values of the loop control variable(s).

3. Inside the loop, the value(s) of the loop control variable(s) must be changed to ensure that the loop condition becomes `false` at some stage.

Very often it is not necessary to initialise the control variable(s) of a `do..while` loop before the loop, since its value can be set in the body of the loop. We therefore only have two rules for a `do..while` loop:

1. Inside the body of the loop, the value(s) of the loop control variable(s) must be changed to ensure that the loop condition becomes `false` at some stage.
2. The condition of the `do..while` loop must test the values of the loop control variable(s).

Counter-driven `while` loops

In a counter-driven loop the number of iterations is determined by a variable that keeps count of the number of times the loop is executed. Before the loop, the counter is given an initial value and is then incremented on each iteration. When this variable reaches a certain value, the loop ends. To use a counter-driven loop, the exact number of times the loop should execute should be known before the loop commences.

Example:

```
count = 1;
while (count <= 10)
{
   cout << "Iteration number " << count << endl;
   count++;
}
```

Accumulation of a sum

We often use `while` loops to accumulate the sum of a sequence of values. In such a loop some variable acts as an accumulator. This variable is initialised to 0 before the loop and then, inside the loop, the values of which the sum must be calculated, are added to it one-by-one.

Example 1:

```
count = 1;
sum = 0;
while (count <= 10)
{
   cin >> value;
   sum += value;
   count++;
}
```

Similarly, we can calculate the product of a sequence of values using a `while` loop. In this case, however, the accumulator variable must be initialised to 1 and not to 0. (See if you can predict what will happen if we initialise product to 0 and not 1 in the example below.)

Example 2:

```
count = 1;
```

```
product = 1;
while (count <= 10)
{
   cin >> value;
   product *= value;
   count++;
}
```

Sentinel-driven loops

Sometimes we use a loop to input a sequence of values without knowing how many values appear in the list. One way to handle such a loop is to ask the user to enter some unique value after the last value has been given. This unique value is called the *sentinel*.

Example:

```
product = 1;
cin >> value;
while (value != 0)
{
   product *= value;
   cin >> value;
}
```

The loop executes until the value 0 is input. In other words, 0 is the sentinel.

Situation-driven loops

The number of times a loop executes is not always determined by a counter or a sentinel. A situation-driven loop ends when a specific situation is reached. It also contains one or more loop control variables, but these are not counter variables.

Example:

```
cin >> x;
while (x != 1)
{
   if (x % 2 == 1)
      x = x * 3 + 1;
   else
      x = x / 2;
   cout << x << endl;
}
```

In this example, the loop ends when the value of `x` becomes 1. It is impossible to say in advance how many times this loop will execute. You might like to draw variable diagrams or write a short program that includes this code to see how the value of `x` changes. It's quite interesting!

The three rules for `while` loops also apply to situation-driven loops: The loop control variable must be initialised, it must be tested in the condition, and it must be changed inside the loop.

Loops that never execute

It is possible that the statements in a `while` loop never execute. Assume, for example, the loop control variable `x` is initialised to 20 before the loop, and the loop condition is `x <= 10`. The condition is `false` when the loop is first encountered and therefore the loop statements do not execute.

A boolean loop control variable

A `while` loop can also be controlled by a boolean variable. The same rules apply here as with the other `while` loops: The boolean variable must be initialised before the loop, and inside the loop it should be changed. For example,

Example:

```
again = true;
while (again)
{
   //Do something
   cout << "Do you want to do it again (Y/N)? ";
   cin >> answer;
   if (answer == 'n' || answer == 'N')
      again = false;
}
```

Initially, `again` is `true`. The `//Do something` comment represents any C++ statement(s). The other statements inside the loop ask the user to type `Y` or `N`. Only when the user types `N` or `n` will `again` get the value `false`. The loop will then end.

`Do..while` vs `while` loops

A `do..while` loop is suitable when we are not certain how many times the loop should execute, but we are sure it must be executed at least once. If there is a possibility that the statements will not be executed at all, use a `while` loop.

Block variables

There are three disadvantages of using block variables in the body of a loop:

(i) They are inaccessible after the loop because they are destroyed at the end of the block.

(ii) They make a program inefficient because every time a block variable declaration is encountered, memory has to be set aside for the variable. And every time the end of the block is reached (i.e. the end of the body of the loop) the variable is destroyed.

(iii) They hide variables with the same name declared in the enclosing function. They are a common cause of logic errors in a program.

Exercises

Exercise 14.1

Write a program that asks the user how many people participated in a survey. It then inputs the height of each of the people and calculates the average height.

Exercise 14.2

Write a program that maintains a simple cheque account. It first asks for a balance and then for a sequence of transactions. Deposits are given as positive values and cheques written as negative values. After each transaction, the program should display the new balance. The transactions end when the value 0 is entered.

Exercise 14.3

Write a program to display the words of the following song:

```
There were 10 in the bed
And the little one said:
"Roll over, roll over!"
So they all rolled over,
And one fell out,
There were 9 in the bed
And the little one said:
:
:
There was 1 in the bed
And the little one said:
"Good night!"
```

Use a `while` loop.

Exercise 14.4

Write a program that inputs a list of yearly salaries and counts how many of these exceed R 100 000. The number of salaries in the list is not known and there is no sentinel that indicates the end of the list. The program should repeatedly ask the user whether there are more values to be input. A message should eventually be displayed that indicates what percentage of the salaries entered is higher than R100 000. Use a `do..while` loop in the program.

Exercise 14.5

What will the output of the following program be? Explain your answer.

```
//Variable scope
#include <iostream>
using namespace std;

int i = 23;

int main( )
{
   int i = 5;
   int j = 10;
   while (j > 0)
   {
      int i = j*j;
      j--;
   }
   cout << "The value of i is " << i << endl;
   return 0;
}
```

Lesson 15

For loops

Purpose of this lesson

In the previous lesson we discussed, amongst other things, counter-driven loops. C++ offers a simpler structure for this kind of loop, namely the `for` statement. In this lesson we discuss simple `for` loops. In the next lesson we will look at nested `for` loops.

Activity 15.a

Write a program to calculate the compound interest on an amount of money at a given interest rate, for a number of years. After inputting an amount of Rands, an interest rate and a number of years to invest the money, the program must calculate and display the amount plus compound interest after each year.

Here is an example of what should appear on the screen while the program is running:

```
Enter an amount of Rands: 1000.00
Enter the interest rate: 12.0
Enter the number of years: 4
After 1 years the amount will be 1120.00
After 2 years the amount will be 1254.40
After 3 years the amount will be 1404.93
After 4 years the amount will be 1573.52
```

Use a `for` loop.

Test yourself

Think about it. You should be able to deal with the repetition required in this program using a `while` loop. The first subactivity below shows what a `for` loop looks like, and you should be able to complete the activity after it.

Subactivity 15.a.i

`For` loops are very similar to counter-driven `while` loops. Below is a counter-driven `while` loop and the `for` loop that is equivalent to it. Both loops display the following 10 lines of output:

```
Line 1
Line 2
:
```

```
    Line 10
```

```
int i = 1;
while (i <= 10)
{
   cout << "Line " << i << endl;
   i++;
}
```

```
for (int i = 1; i <= 10; i++)
   cout << "Line " << i << endl;
```

Compare the two loops and write down their similarities and differences in your study notebook.

Subactivity solution

SIMILARITIES:

1. *Both loops use the counter variable i.*
2. *In both loops the value of i runs from 1 to 10 (the loops each execute 10 times).*
3. *Each includes a cout statement that produces the required output - these statements are identical.*

DIFFERENCES:

1. *Before the while loop, i is declared and initialised whereas with the for loop, this is done in the for statement itself.*
2. *Inside the while loop, i's value is incremented by 1. Once again this is done in the for statement itself.*
3. *We need braces for the two statements in the body of the while loop, but we don't need them for the for loop because there is only one statement in its body.*

Discussion

What we learn from this subactivity is that the three rules that are so important when using `while` loops also apply to `for` loops, except that they are all included in the `for` statement itself. Consider the following loop:

```
cout << "How old are you? ";
cin >> age;
for (int i = 1; i <= age; i++)
   cout << "Happy Birthday!" << endl;
```

In this loop, `i` is the control variable. It is initialised to 1, tested whether it is equal to `age`, and increased by 1 each iteration. For each value of i, the `cout` statement is executed.

Subactivity 15.a.ii

Write a program that displays a times table for any number. For example, say the user enters 7, the program should display

```
The 7 times table is:
7 x 1 = 1
7 x 2 = 2
:
7 x 12 = 84
```

Your program should use a `for` loop.

Subactivity solution

The program below uses a `for` loop with a counter variable whose value ranges from 1 to 12. Inside the `for` loop there is a `cout` statement that displays the next multiple of the number the user chose.

```
//Displays a times table for any number
#include <iostream>
using namespace std;

int main( )

{
   int number, multiple;

   cout << "Enter a number: ";
   cin >> number;

   cout << "The " << number << " times table is:" << endl;
   for (int i = 1; i <= 12; i++)
   {
      multiple = number * i;
      cout << number << " x " << i << " = " << multiple << endl;
   }

   return 0;
}
```

Since the body of the `for` loop contains more than one statement, we use braces. Note that we indent the statements to the right to show clearly that they are part of the body of the `for` loop.

Note how we use the control variable in the body of the loop, both in the assignment statement and in the output statement.

Subactivity 15.a.iii

Write a program to add interest to an amount of money, but only for one year. (In other words, the program should not need a loop.) Your program may only have two variables: one for the amount of money and one for the interest rate.

Subactivity solution

```
//Interest on an amount of money for one year
#include <iostream>
```

```
using namespace std;

int main( )
{
   float amount, rate;

   //Input the amount and the interest rate
   cout << "Enter an amount in Rands: ";
   cin >> amount;
   cout << "Enter the interest rate: ";
   cin >> rate;

   //Set the output format

   cout.setf(ios::fixed);
   cout.precision(2);

   //Calculate and add interest
   amount = amount + amount * rate / 100;
   cout << "After 1 year the amount will be " << amount << endl;

   return 0;
}
```

The reason why we required that you only use two variables is to make things easier for the solution to the main activity.

Fix the above program to provide the solution to the main activity. All you need to do is to add the necessary steps for a `for` loop.

Activity solution

```
//Compound interest on an amount of money for a number of years
#include <iostream>
using namespace std;

int main( )
{
   float amount, rate;
   int years;

   //Input amount, interest rate and number of years
   cout << "Enter an amount in Rands: ";
   cin >> amount;
   cout << "Enter the interest rate: ";
   cin >> rate;
   cout << "Enter the number of years: ";
   cin >> years;

   //Set the output format
   cout.setf(ios::fixed);
   cout.precision(2);
```

```
   //Calculate and add interest for each year
   for (int i = 1; i <= years; i++)
   {
      amount = amount + amount * rate / 100;
      cout << "After " << i << " years the amount will be " << amount << endl;
   }

   return 0;
}
```

Activity 15.b

A simple rule to determine whether a year is a leap year is to test whether it is a multiple of 4.

Write a program to input a year number and a number of years, and then to determine and display which of those years were (or will be) leap years. Here is an example of the user interaction:

```
What year do you want to start with? 1994
How many years do you want check? 8
1994 isn't a leap year
1995 isn't a leap year
1996 is a leap year
1997 isn't a leap year
1998 isn't a leap year
1999 isn't a leap year
2000 is a leap year
2001 isn't a leap year
```

(In fact, the above rule for determining whether a year is a leap year is an over-simplification. Years that are multiples of 100 but not multiples of 400 are not leap years. For the purposes of this program, we use the simple rule, however.)

> **Test yourself**
>
> With the knowledge you acquired from the previous activity, you should be able to write the program very easily. We give an alternative solution (probably not the first one you'd think of) to show an alternative way of using a `for` loop.

Subactivity 15.b.i

Write a program as a complete solution to this problem now. Your program should input the two values, and then use a `for` loop to repeat the necessary statements from 1 to the numbers of years required. You'll need an `if` statement in the body of the loop to determine whether a year is a leap year, and display the appropriate message.

Subactivity solution

```
//Determines which of a sequence of years are leap years
#include <iostream>
using namespace std;

int main( )
{
   int start, many;

   cout << "What year do you want to start with? ";
   cin >> start;
   cout << "How many years do you want check? ";
   cin >> many;

   for (int i = 0; i < many; i++)
   {
      if ((start + i) % 4 == 0)
         cout << start + i << " is a leap year" << endl;
      else
         cout << start + i << " isn't a leap year" << endl;
   }

   return 0;
}
```

You quite possibly declared an additional variable, say `year`, and stored the value of `start + i` in it at the beginning of the loop to save having to do the calculation `start + i` over and over again. If you did so, well done! We show another way of arranging things below, but before we do so, we discuss a common error when using `for` loops.

Subactivity 15.b.ii

What will the output of the following code be?

```
int total = 0;
for (int i = 10; i <= 20; i++)
   total++;
cout << total << endl;
```

Subactivity solution

If you think the answer is 10, you had better think again.

It is very important to note that a `for` loop is executed once for the initial value of the counter variable (in this case 10), once for the final value (in this case 20), and once each for all the values in between (11, 12, 13, 14, 15, 16, 17, 18 and 19).

Discussion

The above subactivity illustrates a common mistake when using counter controlled loops, namely a *fence-post error*. Say some fence posts are 10m apart, and you count 10 fence posts, what distance does that represent?

The answer is 90m, not 100m as one would immediately think. This is because the distances are between the posts, so if you have 10 fence posts, you've got 9 spaces between them.

A similar error can occur in programming, especially when dealing with `for` loops. You should always carefully check the initial and final values of the control variable and make sure that the loop will be executed the exact number of times you want it to be repeated.

Subactivity 15.b.iii

Now change the program you wrote for Subactivity 15.b.i so that the control variable of the `for` loop runs from the first year number to the last. Beware of a fence-post error!

This will give the solution to the main activity.

Activity solution

Our solution looks as follows:

```
//Determines which of a sequence of years are leap years
#include <iostream>
using namespace std;

int main( )
{
   int start, many;

   cout << "What year do you want to start with? ";

   cin >> start;
   cout << "How many years do you want check? ";
   cin >> many;

   for (int i = start; i < start + many; i++)
   {
      if (i % 4 == 0)
         cout << i << " is a leap year" << endl;
      else
         cout << i << " isn't a leap year" << endl;
   }

   return 0;
}
```

Here it would have been quite acceptable, perhaps even preferable, to call the control variable `year` rather than `i`, since it represents something more that merely a counter. It is the year number we are currently working with.

Note the final value specified for terminating the `for` loop. If we had used a less-than-or-equals sign `<=` instead of the less-than sign `<`, we would have had to use the expression `i <= start + many - 1` to avoid a fence post error. Check this to see whether you agree.

Activity 15.c

In your study notebook, answer the following questions:

1. What are the values of variables declared in the body of a `for` loop after the loop has ended?
2. What is the value of the control variable after a `for` loop has ended?
3. What happens if you change the value of the control variable inside the body of a `for` loop?
4. What happens if you change the values of other variables used to specify the final value of the control variable in the body of a `for` loop?

> **Test yourself**
>
> We recommend that you take a guess at answering each of these questions before working through the subactivities. Then check your answers with those given in the solution to this activity.

Subactivity 15.c.i

The answer to the first question is the same as for `while` loops. If you can't remember, go back to Activity 14.f.

Subactivity 15.c.ii

Write a short program to output the value of the control variable after a `for` loop. Test the following two cases: (i) Declare the counter variable in the initialisation part of the `for` loop. (ii) Declare the counter variable before the `for` loop.

Subactivity solution

Here is our program:

```
//To test the value of the control variable after a for loop
#include <iostream>
using namespace std;

int main( )
{
   int i;
   for (i = 10; i <= 20; i++)
```

```
      cout << "i = " << i << endl;
   cout << "Final value: " << i << endl;

   return 0;
}
```

You should have discovered that the control variable is inaccessible after the `for` loop if it is declared in the initialisation part of the loop. It is only accessible if it is declared before the `for` loop. In other words, it's like a block variable, except that it is not created and destroyed on each repetition of the loop.

After the loop, the value of the control variable is 1 more than the final value specified in the condition part of the `for` loop. (If the loop decrements the value of the control variable, it will be 1 less than the final value.)

Subactivity 15.c.iii

What is the output of the following program fragment?

```
int j = 5;
for (int i = 1; i <= j; i++)
{
   cout << j << ' ';
   j--;
}
```

Subactivity solution

The output is:

```
5 4 3
```

This illustrates an important principle regarding the `for` structure: The final value of the counter variable can be changed during execution of the loop. The variables that determine the final value (such as `j` above) can be modified inside the loop, and this will affect the number of times the loop will execute.

In fact, it is considered bad programming practice to change the values of a variable that acts as the final value of a `for` loop since it makes it difficult to predict how many times the loop will be executed. In general, we use a `for` loop when we can determine beforehand how many times the loop must be executed, and a `while` loop if we can't.

Subactivity 15.c.iv

What will the output of the following two loops be?

A.

```
for (int k = 1; k <= 4; k++)
{
   cout << k << endl;
   k--;
}
```

B.

```
for (int k = 1; k <= 4; k++)
{
   cout << k << endl;
   k++;
}
```

Subactivity solution

Although the initial value of the counter variable is set before the first iteration, this does not mean that the number of iterations is determined then. It is possible to change the value of the counter variable in the body of the loop and in so doing alter the number of times the loop will execute.

The output for A is:

```
1
1
1
1
:
```

infinitely many times.

The `for` loop initially determines that `k`'s value will range from 1 to 4. During each iteration, the `for` loop increments `k` by 1, but inside the body of the loop, `k`'s value is also decremented by 1. The result is that the value displayed for `k` remains constant and the final value, 4, is never reached. The loop thus runs forever.

The output for B is:

```
1
3
```

In this case, not only does the `for` loop increment `k` by 1, but `k` is also incremented in the body of the loop. This means that the final value 4 is reached after only two iterations.

Discussion

These results are possibly contrary to your expectations. It illustrates an important principle regarding the `for` loop: The statements inside the `for` loop may use the counter variable, but if the value of the counter variable is modified inside the body of the loop, this might give rise to undesired or unexpected consequences.

In general, it is considered bad programming practice to modify the value of the counter variable in the body of a `for` loop. If you ever feel that you need to do so for a particular program, use a `while` loop. This means that the number of iterations could not be determined before the for loop commenced. You should always use a `while` (or a `do..while` loop) if it is not possible to determine the number of iterations before the loop commences.

We have now covered all the aspects needed to answer the four questions in the main activity. Before looking at our solution below, read the answers you wrote down previously and revise them if necessary.

Activity solution

Here are the answers (we repeat the four questions):

1. What are the values of variables declared in the body of a `for` loop after the loop has ended?

The same applies as with while loops: All variables declared in the body of a for loop are inaccessible after the loop. They are created each time the declaration is executed and destroyed when the end of the block is reached.

2. What is the value of the control variable after a `for` loop has ended?

   If the control variable is declared in the initialisation part of the for loop, it is inaccessible after the loop. This is similar to variables declared in the body of the loop, except that the control variable is not destroyed and recreated each time the loop is repeated. It is created once in the initialisation part and destroyed when the loop terminates.

3. What happens if you change the value of the control variable inside the body of a `for` loop?

   If you change the value of the control variable inside the body of a for loop, the number of repetitions of the loop will be affected. The loop may be repeated less or more times than expected, or may even go into an infinite loop.

4. What happens if you change the values of other variables used to specify the final value of the control variable in the body of a `for` loop?

   This can also affect the number of repetitions of the loop. It may be repeated more or less times than expected, or may go into an infinite loop.

Activity 15.d

Consider the following program and answer the questions below it:

```
//What does it do?
#include <iostream>
using namespace std;

int main( )
{
   float a, x;
   const float TOLERANCE = 0.00001;

   cout << "Enter a floating point number greater than 1: ";
   cin >> a;

   for (x = a; x*x*x - a > TOLERANCE; x = (2*x + a/(x*x))/3)
      cout << x << endl;

   cout << "The answer is " << x << endl;

   return 0;
}
```

(i) Explain what the above program does. (Type in the program and run it a few times to check that it works, and to see what output it gives.)

(ii) Apart from values less than 1, there are other values for which the program does not work correctly. Change the program so that it works for all input values greater than 1 and is more readable.

Test yourself

In your study notebook, write down what you think the program does. Don't merely translate the code into English, eg. "The program declares a variable called `a` and inputs a value for it; it then enters a `for` loop ... " etc. Try to explain what the purpose of the program is, i.e. what it is meant to calculate; what problem the programmer was solving.

We provide one subactivity to help you work out what the program does (if you can't work it out yourself). Then we give a few subactivities to show you how to change the code to make it more readable.

Subactivity 15.d.i

Change the program by replacing the `for` loop with the following:

```
for (x = a; x*x - a > TOLERANCE; x = (x + a/x)/2)
   cout << x << endl;
```

Run the program a few times again and see if you can guess what it calculates. Also, investigate which values it doesn't work for.

Subactivity solution

This (changed) program calculates the square root of a number. In other words, it inputs a value for `a` and then calculates what number, when multiplied by itself, will give `a`.

Note that it doesn't calculate the exact square root, it calculates an approximation. Each repetition of the loop, variable `x` contains a value closer and closer to the exact square root. The constant `TOLERANCE` determines the level of accuracy of the final answer.

Apart from the fact that the program doesn't work for negative numbers, if the input value is greater than some threshold (eg. 275) it goes into an infinite loop.

Discussion

You should have been surprised by the strange syntax of the `for` loops given above. In particular, the control variable (in this case `x`) is not an integer, it is a floating point variable. Secondly, it is initialised to a floating point value obtained from another variable (i.e. `a`) rather than a literal integer value (as we have used up to now). Thirdly, the condition for terminating the `for` loop does not test whether the control variable is less than or greater than some value, but rather whether the difference between the control variable squared and the original value is less than some tolerance value. Finally, the part of the `for` loop that changes the value of the control variable does not simply increment or decrement it, it uses a complex expression to assign a new value to it.

Below, we will discuss the pros and cons of using such strange ways of initialising, testing and changing the value of the control variable in a `for` loop. The point of this subactivity (and the main activity) is that it can be done.

Subactivity 15.d.ii

One way to make the above program work for more input values is to change the tolerance value. (You might like to change the value of `TOLERANCE` to see how it affects the threshold.) A better way is to limit the number of repetitions of the loop. As you have no doubt noticed, in most cases the value of `x` very quickly converges to a value close to the exact square root.

Change the `for` loop so that it repeats a maximum of 20 times. In other words, if it reaches a value within the tolerance level before that, it should terminate earlier.

Subactivity solution

What we really need is two control variables: the existing variable `x` and a counter variable (say `i`). We need to ensure that both are initialised, both are tested and both are changed in the loop. Here is one solution:

```
int i = 0;
for (x = a; x*x - a > TOLERANCE && i < 20; x = (x + a/x)/2)
{
   cout << x << endl;
   i++;
}
```

Note: We can't initialise two variables in the initialisation part of a `for` loop, nor can we use two statements to change them in the final part. We therefore have to initialise one of them before the loop, and change one of them in the body of the loop. (We can test both control variables in the condition part of the `for` loop by using the `&&` operator.)

Here is another method that some C++ programmers would use:

```
int i = 0;
for (x = a; x*x - a > TOLERANCE; x = (x + a/x)/2)
   if (i++ > 20)
      break;
   else
      cout << x << endl;
```

We don't like this solution for a number of reasons. Firstly, it is longer and more complicated than the previous solution. Secondly, is uses a `break` statement (as we use in a `switch` statement) to terminate (i.e. jump out of) the loop prematurely. We prefer not to do this as it decreases the readability of a program. In particular, the condition for termination of the loop is not clear from this code: If you look at the `for` loop, it appears that it will only terminate when the difference between `x*x` and `a` is less than `TOLERANCE`.

Our final solution is to turn the whole loop into a `while` loop:

```
int i = 0;
x = a;
while (x*x - a > TOLERANCE && i < 20)
{
   cout << x << endl;
   x = (x + a/x)/2;
   i++;
}
```

We hope that you agree that this is the most satisfactory solution. It might be a bit longer than our first solution, but now it is clear that there are two control variables, that they are both initialised before the loop, tested in the condition of the loop, and changed in the body of the loop.

You should be able to answer the three questions for the problem of the main activity now. For part (iii), get rid of the `for` loop and replace it with a `while` loop.

Activity solution

Like the simplified program that calculated the square root of a number, the program of the main activity calculates the cube root of a number. In other words, it inputs a number and calculates a value which, when multiplied by itself three times, gives the number.

The original program also suffers from the problem of going into an infinite loop when the input number exceeds some threshold value. For this reason, we also introduce an additional control variable that restricts the loop to execute a maximum of 20 times:

```
//Calculate the cube root of a number
#include <iostream>
using namespace std;

int main( )
{
   float a, x;
   const float TOLERANCE = 0.00001;

   cout << "Enter a floating point number greater than 1: ";
   cin >> a;

   int i = 0;
   x = a;
   while(x*x*x - a > TOLERANCE && i < 20)
   {
      cout << x << endl;
      x = (2*x + a/(x*x))/3;
      i++;
   }

   cout << "The answer is " << x << endl;

   return 0;
}
```

Discussion

Both the main activity and the subactivities illustrate the so-called *Babylon method* for determining the square root and cube root of a number. Note that this method is NOT important for the purposes of this guide. What is important is that a `for` loop can look quite different to the `for` loops that we have seen so far. In particular, the parts for initialisation, testing and changing of the control variable

can contain complex statements and/or conditions. Furthermore, the body of a `for` loop can contain a `break` statement to cause the loop to terminate prematurely.

Although we do not encourage the use of these techniques in your C++ programs, we have included this activity so that you will not be surprised when you see C++ programs written by people who use these tricks in their programs. In fact, there are some situations where it could be acceptable to use a `for` loop rather than a `while` loop, even when it isn't strictly a counting loop, or to use a `break` statement to jump out of a loop. The question to ask is whether it increases or decreases the readability of the code.

Important points in this lesson

Programming concepts

The `for` loop is a counter-driven loop. The most common structure of a `for` loop looks as follows:

```
for (int Counter = InitialValue; Counter <= FinalValue; Counter++)
   Statement;
```

Counter is the loop control variable that the `for` statement initialises to *InitialValue*, and increases by 1 until the value reaches *FinalValue*. *Statement* is executed for each value of *Counter*.

In this form, *Counter* is an integer variable, and *InitialValue* and *FinalValue* are variables, literals or expressions of the same type. If *InitialValue* is greater than *FinalValue*, the loop is never executed. If the two values are equal, the loop executes only once.

We can get a `for` loop to count down by decrementing *Counter* in the last part of the `for` statement. The condition part of the `for` loop should be changed accordingly. In other words, the relational operator that tests whether *Counter* has reached *FinalValue*, will need to be changed.

Any variables declared in the body of a `for` loop are inaccessible after the loop.

If the counter variable is declared in the initialisation part of a `for` loop, it is inaccessible after the `for` loop. If it is declared before the `for` loop, its value will be 1 more than the final value specified in the condition part of the `for` loop (or 1 less if the control variable is decremented).

The control variable need not be of integer type, and it can be initialised with any (complex) assignment statement. The condition for termination of a `for` loop can also be a complex condition involving numerous variables. Also, the third part of the `for` statement need not be a simple increment or decrement statement. It can in fact be any C++ statement. The body of a `for` loop can contain a `break` statement. This will cause the loop to terminate prematurely.

Programming principles

Use a `for` loop when you can determine the number of iterations before the loop commences (as opposed to using a `while` loop when you can't determine the number of iterations before the loop commences).

The body of the `for` loop can use the counter variable, but changing the value of the counter variable in the body of the loop can give rise to unwanted side-effects. In general it is considered poor programming practice to change the value of the counter variable inside the body of a `for` loop.

Changing the values of any variables used in the condition (for determining loop termination) in the body of the loop will affect the number of iterations. In general it is considered poor programming practice to

change such variables inside the body of a `for` loop.

Any relational operator can be used to test whether the counter variable has reached the final value, but it is dangerous to use `!=` because if the counter variable manages to jump over the exact final value, the loop will not terminate. Rather use `<`, `<=`, `>` or `>=` as appropriate.

Although the `for` loop is very flexible in terms of how the control variable is initialised, tested and changed, we prefer to stick to the simple structure given at the beginning of the **Programming concepts** section above. If you do not need a counting loop, or you need to use two control variables, or you need to terminate a `for` loop prematurely (i.e. before the counter has reached its final value) rather use a `while` loop.

Exercises

Exercise 15.1

Write a program that calculates and displays the sums of all the odd numbers up to a number `n`. The user must supply a value for `n`, and the program must display all the partial sums up to `n` (including `n` if `n` is odd).

Exercise 15.2

Write a program that uses a `for` loop to display all the characters with ASCII values from 32 to 255 on the screen.

Exercise 15.3

Write a program that uses a `for` loop to display the ASCII values of all the upper-case letters of the alphabet. Display each letter with its ASCII value on a separate line.

Exercise 15.4

Redo Exercise 14.3, but this time use a `for` loop rather than a `while` loop.

Lesson 16

Nested loops

Purpose of this lesson

Like `if` statements, loops can also be nested. In this lesson we look at loops that form part of the statements of another loop. Each time the outer loop is executed, the inner loop is executed right from the start. That is, all the iterations of the inner loop are executed with each iteration of the outer loop.

Activity 16.a

Write a program that simulates the following game:

A coin is tossed repeatedly until the first head appears. Suppose it takes N tosses to get the first head. A payoff of R2^N is then made. For example: Tail, Tail, Head pays R8.00, that is 2^3, since there were 3 tosses. The program should play the game repeatedly. It starts by asking the user how many times the game will be played. The tosses of the coin should be entered from the keyboard: `H` for "heads" and `T` for "tails". When a game is over, the program should calculate the payoff for the game and display it. When all the games have been played, the program should calculate and display the average of the payoffs for all the games.

Test yourself

Use nested loops in this program. If you are not sure how to use them, or which loops to use, work through the subactivities below.

Subactivity 16.a.i

Consider the following simple program:

```
//Simple loop
#include <iostream>
using namespace std;

int main( )
{
   for (int j = 1; j <= 5; j++)
      cout << j;
   cout << endl;

   return 0;
}
```

This program displays the line:

```
12345
```

Change the program so that it displays this line 5 times. It should give the following output:

```
12345
12345
12345
12345
12345
```

Subactivity solution

We must change the existing program so that it repeats 5 times what it currently does once. To do this, we merely enclose the existing statements in a `for` loop that executes 5 times.

```
//Nested loop
#include <iostream>
using namespace std;

int main( )
{
   for (int i = 1; i <= 5; i++)
   {
      for (int j = 1; j <= 5; j++)
         cout << j;
      cout << endl;
   }

   return 0;
}
```

What we have here are nested `for` loops. The inner loop's counter, `j`, goes from 1 to 5 for each value of `i` (the outer loop's counter). When `i` is 1, all 5 iterations of the inner loop are executed; when `i` is 2, all the iterations of the inner loop are executed again, and so on. After the inner loop has displayed the 5 values, the program skips to the next line with the `cout << endl;` statement before the next iteration of the outer loop starts.

Subactivity 16.a.ii

What will be displayed by the following program segment?

```
   for (int i = 1; i <= 5; i++)
   {
      for (int j = 1; j <= i; j++)
         cout << j;
      cout << endl;
   }
```

Subactivity solution

The output looks as follows:

```
1
12
123
1234
12345
```

Since the value of `j` goes from 1 to `i`, and `i`'s value changes all the time, the inner loop does not always repeat the same number of times. The number of iterations of the inner loop depends on the value of `i`.

When `i` is 1, the inner loop executes once, since `j` then goes from 1 to 1. 1 is the only value displayed by the inner loop.

When `i` is 2, the inner loop executes twice, since `j` then goes from 1 to 2. 1 and 2 are the only values the inner loop displays.

:

When `i` is 5, `j` goes from 1 to 5, and all 5 values are displayed by the inner loop.

Subactivity 16.a.iii

In Activity 14.h you were asked to solve the following problem:

> The number of digits in an integer can be determined by counting the number of times the integer can be divided by 10 before the quotient is 0. Write a program to count the number of digits in a given integer. Your program should use a `do..while` loop.

Write the program again, but this time repeat the whole process until the user enters the value 0. That is, the user may enter any number of values, and for each value the program must report the number of digits it consists of. Use only `do..while` loops.

Subactivity solution

```
//Calculates the number of digits in a number
#include <iostream>
using namespace std;

int main( )
{
   int number, num, count;

   do
   {
      //Prompt the user for a number
      cout << "Enter an integer (0 to end): ";
      cin >> number;
      num = number;
```

```
      //Determine the number of digits in the number
      if (num != 0)
      {
         count = 0;
         do
         {
            count++;
            num /= 10;
         } while (num != 0);

         cout << number << " contains " << count << " digit(s)" << endl;
      }

   } while (number != 0);

   return 0;
}
```

To repeat the whole process, we included all the statements in the program in a `do..while` loop. The program inputs a number and if the number is not 0, it determines the number of digits in it. It then inputs the next value, until the input value is 0.

Subactivity 16.a.iv

Adapt the program you wrote for the previous subactivity to use a `while` loop as the outer loop.

Subactivity solution

```
//Calculates the number of digits in a number
#include <iostream>
using namespace std;

int main( )
{
   int number, num, count;

   //Prompt the user for a number
   cout << "Enter an integer: ";
   cin >> number;

   while (number != 0)
   {
      num = number;

      //Determine the number of digits in the number
      count = 0;
      do
      {
         count++;
         num /= 10;
      } while (num != 0);

      cout << number << " contains " << count << " digit(s)" << endl;
```

```
      //Input another number
      cout << "Enter another integer (0 to end): ";

      cin >> number;
   }

   return 0;
}
```

Discussion

Since a `while` loop is a pretest loop, we have to input the first value outside the loop so that number has a value when the condition is tested the first time. Note that we do not need the `if` statement that was included in the `do..while` loop in the previous subactivity. We had to include it there otherwise that program would also have reported the length of the sentinel. The `while` loop's condition is tested right after the value is input, so the loop will end immediately if the input is 0.

Subactivity 16.a.v

During a dog show, a group of dogs are judged by 10 people. Each judge awards a score out of 10 for each dog. The scores of the different judges are added together to give a score out of 100 for each dog. Write a program that does the following for each dog: Input the dog's entry number, as well as the 10 scores for that dog. Then calculate the total score out of 100. The program ends when 0 is input for a dog's entry number.

You will need to use nested loops for this program. Before you write it, decide which loops to use.

Subactivity solution

For the outer loop, we use a `while` loop that continues to execute until the sentinel value 0 is entered. Inside the `while` loop we have a `for` loop that inputs the 10 scores for the dog and calculates the total score. Finally, we display the score and input the next dog's entry number.

```
//Calculates scores for dogs at a dog show
#include <iostream>
using namespace std;

int main( )
{
   int dogNo, score, totalScore;

   cout << "DOG SHOW" << endl << endl;
   cout << "Enter the first dog's number: ";
   cin >> dogNo;

   while (dogNo != 0)
   {
      //Input and calculate total for one dog
```

```
      totalScore = 0;
      cout << "Enter the 10 scores for dog no " << dogNo << endl;
      for (int i = 1; i <= 10; i++)
      {
         cin >> score;
         totalScore += score;
      }
      cout << "Dog no " << dogNo << " scored " << totalScore << endl << endl;

      cout << "Enter the next dog's number (0 to stop): ";
      cin >> dogNo;
   }

   return 0;
}
```

The advantage of inputting a value once before the loop and once at the end of the body of the `while` loop is that we can display different prompt messages. Compare the different messages in the above program.

Subactivity 16.a.vi

Consider the problem statement of the main activity again:

> A coin is tossed repeatedly until the first head appears. Suppose it takes N tosses to get the first head. A payoff of R2^N is then made. For example: Tail, Tail, Head pays R8.00, that is 2^3, since there were 3 tosses. The program should play the game repeatedly. It starts by asking the user how many times the game will be played. The tosses of the coin should be entered from the keyboard: `H` for "heads" and `T` for "tails". When a game is over, the program should calculate the payoff for the game and display it. When all the games have been played, the program should calculate and display the average of the payoffs for all the games.

How many loops will be needed in the program to solve this problem, and what sort of loop should each be (`while`, `do..while` or `for`)? Which loops will be nested in the other(s)?

Subactivity solution

Note that the word repeatedly appears twice in the problem statement. This is an indication that we should use (at least) two loop structures. Because the user specifies in the beginning how many times the game should be repeated, one loop should be a for loop. The other repetitions involve tossing the coin until it shows heads. We don't know how many tosses it will take, and therefore need to use an situation-driven loop structure for this task. This can either be a while loop or a do..while loop. We know there will be at least one toss, so this seems to indicate that we should use a do..while loop. However, calculations only need to be made if a tail is tossed, which means that the calculations might not need to be done at all (if heads is tossed the first time). So either choice will work.

Which loop is the outer loop and which the inner loop? Every time the game is played, the coin is tossed until it shows heads. The outer loop is thus

```
   for (int i = 1; i <= noOfGames; i++)
```

and the inner loop

```
while (toss == 'T')
{
   :
}
```

or

```
do
{
   :
} while (toss == 'T');
```

In the inner loop, we need to simulate the tossing of the coin. We will do this by inputting `T` or `H` from the keyboard. We will also need to count the number of tosses, N, because the amount to be out paid depends on that. This amount is calculated as 2^N. The program therefore also needs to calculate 2*2*2*..., N times, so we need another loop! Since the number of repetitions can be determined before this loop commences, we will use a `for` loop.

This computation must take place after the inner loop has finished, but still inside the outer loop.

Now try to write the whole program for the main activity. If you feel that it's still too big to write all at one go, first write the program to play the game only once. Then add the functionality to allow it to be played many times.

Activity solution

The following is one way to write the program:

```
//Simulates a gambling game based on tossing a coin
#include <iostream>
using namespace std;

int main( )
{
   int noOfTosses, noOfGames, payoff;
   char toss;
   float totalPayoff, average;

   //Determine the number of games
   cout << "How many games do you want to play? ";
   cin >> noOfGames;

   totalPayoff = 0;
   for (int i = 1; i <= noOfGames; i++)
   {
      cout << "Game no " << i << ":" << endl;

      //Input tosses until head occurs
      cout << "Enter a toss (H or T): ";
      cin >> toss;
```

```
      noOfTosses = 1;
      while (toss == 'T' || toss == 't')
      {
         cout << "Enter another toss (H or T): ";
         cin >> toss;
         noOfTosses++;
      }

      //Calculate and display payoff
      payoff = 1;
      for (int j = 1; j <= noOfTosses; j++)
         payoff *= 2;
      cout << "Payoff: R" << payoff << endl;

      totalPayoff += payoff;
   }

   //Calculate and display average payoff
   average = totalPayoff / noOfGames;
   cout << "Average payment is R" << average << endl;

   return 0;
}
```

We declare `totalPayoff` as a `float` so that floating point division is used when it is divided by `noOfGames`, to give a floating point value for `average`.

Important points in this lesson

Programming concepts

Like `if` statements, loops can be nested. Nested loops consist of an outer loop with one or more loops that form part of its body. With each iteration of the outer loop, the inner loop is executed from the beginning and all its iterations are carried out.

Programming principles

Just because a program needs more than one loop, doesn't mean that they should necessarily be nested. Perhaps they should be executed consecutively.

Exercises

Exercise 16.1

Write a program that calculates the value of y in the equation $y = x^3 - 3x + 1$, for various series of values of x. The program must input a list of start and end values for x, and for each value of x from the start to the end value, it must calculate the values of y. For example, if the start value is 10 and the end value is 20, it should calculate y for each value of x from 10 to 20. For each value of x, the calculated value of y should be displayed together with x on a separate line.

Pairs of start and end values should be input until both are 0.

Exercise 16.2

Write a program that uses nested `for` loops to display the following multiplication table:

```
    1   2   3   4   5   6   7   8   9
1   1
2   2   4
3   3   6   9
4   4   8   12  16
5   5   10  15  20  25
6   6   12  18  24  30  36
7   7   14  21  28  35  42  49
8   8   16  24  32  40  48  56  64
9   9   18  27  36  45  54  63  72  81
```

Part III

Functions

As you have no doubt realised, programs can become very long and complicated. There are various ways of organising code so that it has a logical structure - so that parts of the code that belong together are kept together. One important way is to write and use functions.

Another important purpose of writing functions is for code-reuse. Say you find that you are having to write code that you have written before. By putting the code that you wrote before in a function, it becomes very easy to reuse the code without having to retype it.

Lesson 17

Using functions

Purpose of this lesson

Before we start writing our own functions, we will look at some of the functions that C++ provides in this lesson.

As we will see, many functions are designed for doing arithmetic and mathematical operations, and so it is not surprising that the standard library functions of C++ include operations such as absolute value, square root, power, the trigonometric functions like sine and cosine, logarithmic functions, and a function for producing random numbers. We will not look at all of these, but will start by using the absolute value and random functions.

Activity 17.a

The local school is holding a competition in an effort to raise funds. They want a number guessing game that can work on a computer. Anyone buying a ticket to the competition guesses a number between 1 and 100. The person whose guess is closest to the correct number, wins a prize.

The program must first generate a secret number (randomly). After that, it must ask for each answer given. Whenever an answer is given that is closer than any previous answer, the program must display the message 'This is the best answer so far'. When all the answers have been entered, the user must enter zero (i.e. 0) to show that there are no more answers to come.

A possible test run through the program would look like this:

```
Guess the secret number!
- Enter each guess.
- When there are no more guesses, enter 0.

Enter the first guess: 80
This is the best guess so far.

Enter the next guess: 100

Enter the next guess: 40
This is the best guess so far

Enter the next guess: 0

The secret number was 50
The best guess was 40
```

Test yourself

If you can write a program that does this successfully, congratulations! You should have used the `rand` function to generate a random number between 1 and 100, and the `abs` function to determine the error between a guess and the secret number.

This is quite a complicated program. If you wrote a program that works, go through our solution and see whether there are any points we mention that you might have missed.

If you did not manage to write a program to solve this program, stop and think a bit about how best to set about writing it. As we have said before, it helps to break the program down into simpler parts rather than trying to write the entire program all in one go. So, where should you start writing this program?

There are two main issues in this problem. One is to generate the secret number randomly, and the other is to work out how close the user's guess is to the correct value.

The first few subactivities below show how to use the `rand` function to generate a random integer, and the next group of subactivities explain how to use the `abs` function to work out how close a guess is to the correct value.

Subactivity 17.a.i

Type in the following program and get it to run:

```
//Display 10 random numbers
#include <iostream>
using namespace std;

int main( )
{
   int r;

   for (int i = 0; i < 10; i++)
   {
      r = rand( );
      cout << "The number is " << r << endl;
   }

   return 0;
}
```

After running the program a few times, add the following statements between the declaration of `r` and the `for` loop:

```
   int seed;
   cout << "Enter a seed for the random number generator: ";
   cin >> seed;
   srand(seed);
```

Compile and then run the program a few more times, using different values for `seed` to see what effect it has.

Explain what happens.

Subactivity solution

With the original program, ten random numbers are displayed, but the same sequence is displayed each time. If we want a different sequence, we can seed the random number generator with the `srand` function. However, if we use the same value for `seed`, we get the same sequence of random numbers.

Subactivity 17.a.ii

Now make the following changes to the program:

- Remove the statements you added for the previous subactivity.
- Insert the statement `srand(time(0));` just before the `for` statement.

Compile and run the program a few more times.

Explain what happens.

Subactivity solution

Now each time the program runs, we get a different sequence of random numbers.

Discussion

The `rand` function returns a random (positive) integer (anything from 0 up to 32767).

As stated in the subactivity solution, `rand` will always start with the same number, and produce the same pattern of random numbers. To get `rand` to start with a different number, we have to seed it with the `srand` function.

The `srand` function takes a single integer value as parameter and uses that to seed the random number generator. In our program we used another function called `time` to provide the number to seed the random number generator.

The `time` function returns a value obtained from the system clock. Each time you run the program, `time(0)` will return a different value (because time marches on!) That's how we ensure that the random number generator is seeded with a different number each time the program is executed.

Subactivity 17.a.iii

Now replace the statement `r = rand( );` with `r = rand( ) % 50;` in the final program.

Either predict what the output will be, or run the program with this change and explain the effect of the change.

Subactivity solution

The `%` operator gives the remainder when one number (in this case the large random number generated by `rand`) is divided by another number (in this case 50). The only remainders that can be obtained by this division are 0 up to 49, so the program generates ten random numbers in the range 0 to 49.

Discussion

In general, the way to generate numbers within a given boundary is as follows: Ask yourself "How many different values do I want to generate?" Call this N. So if you want to generate numbers from 0 to 25 for example, then N is 26. Use `%` with `rand` and N, i.e. `rand( ) % 26`, to give remainders of 0 to $N-1$.

Say you want to generate numbers starting with a value other than 0. Then all you need to do is add the smallest value that you want to the expression explained above. So say you want random numbers from 10 to 30. First count how many possible numbers there are - in this case 21 in all (check this for yourself) - and use the expression `rand( ) % 21`. This will give random values in the range 0 to 20. Now add the smallest number to the expression, i.e. `10 + rand( ) % 21`, to shift the range to 10 to 30.

You can always check your expression by thinking about the smallest and largest values:

- The smallest value generated by `rand( ) % 21` is 0, so the smallest value generated by `10 + rand( ) % 21` will be 10.
- The largest value generated by `rand( ) % 21` is 20, so the largest value generated by `10 + rand( ) % 21` will be 30.

Subactivity 17.a.iv

Write a program that generates a secret random number in the range 1 to 100 and asks the user to guess what it is (just once). The program must display the difference between the guess and the secret number.

You should be able to work out the correct expression to get a value in the range 1 to 100 now.

The next problem is to calculate the difference between the guess and the secret value.

As a start, we can subtract the guess from the secret value. However, if the guess is greater than the secret value, we are going to get a negative number. We can either use an `if` statement, or we need a way to ignore the sign. To tell the computer to ignore the sign, we can use the *absolute value function*, namely `abs`. For example, the value of `abs(-3)` is 3, and the value of `abs(3)` is also 3.

Try writing the program for this subactivity now. Use the following steps:

1) Generate the secret number
2) Input the user's guess
3) Subtract the guess from the secret number (to obtain the "error") and store it in a variable
4) Determine the absolute value of this error
5) Display the result

Subactivity solution

```
//One guess for a secret number
#include <iostream>
using namespace std;

int main( )
{
   int secret, guess, error, absError;

   srand(time(0));
   secret = 1 + rand( ) % 100;

   cout << "Guess the secret number: ";
   cin >> guess;

   error = secret - guess;
   absError = abs(error);
   cout << "You were out by " << absError << endl;

   return 0;
}
```

The `abs` function returns a value, so we can assign it to another variable that we have already declared. In the example above, we send the value of error as a parameter to function `abs`, and then assign the value returned by `abs` to the variable `absError`. We can then use `absError` later on in the program.

A second way of writing this program is:

```
//One guess for a secret number
#include <iostream>
using namespace std;

int main( )
{
   int secret, guess, error;

   srand(time(0));
   secret = 1 + rand( ) % 100;

   cout << "Guess the secret number: ";
   cin >> guess;

   error = secret - guess;
   cout << "You were out by " << abs(error) << endl;

   return 0;
}
```

As we said above, the `abs` function returns a value, so we can use this value directly in the `cout` statement as we have done in this version of the program. We don't need the variable `absError` any more. However, as

we said in one of the first lessons of this study guide, we prefer not to do calculations in output statements, but rather to introduce an extra variable.

We can get rid of another variable by changing the call to the `abs` function, however. Instead of having a separate line to calculate the value of error, and then sending this value to the function, we can calculate the error in the function call as follows:

```
//One guess for a secret number
#include <iostream>
using namespace std;

int main( )
{
   int secret, guess, absError;

   srand(time(0));
   secret = 1 + rand( ) % 100;

   cout << "Guess the secret number: ";
   cin >> guess;

   absError = abs(secret - guess);
   cout << "You were out by " << absError << endl;

   return 0;
}
```

Note that function `abs` still takes a single parameter. In this case, the single value required by `abs` is provided by the expression `secret - guess`. Even though this expression contains two variables, the result of the expression is a single integer.

Subactivity 17.a.v

Having worked out how to find the difference correctly, we can now go on to look at how to ask the user to enter more than one guess. Before writing out the program, write out an algorithm to solve this problem in your study notebook, similar to what we did in Subactivity 17.a.iv. (If you need a hint on how to input a series of numbers, it might be worth looking over Lesson 9 again.)

Subactivity solution

One can write out an algorithm like this in different levels of detail, and ours is quite a high level one that does not show much detail. Yours may be more detailed.

1) Generate the secret number

2) Input the first guess

3) If this is the best guess so far, keep a record of it, otherwise ignore it

4) Input the next guess, and loop back to 3)

5) When all the guesses have been entered, display the best guess.

Subactivity 17.a.vi

Based on this algorithm and on the subactivities in this lesson so far, write the program specified for the main activity. Save it and run it before comparing your program with our solution.

Activity solution

In our program, we have included the algorithm as comments. We also give other comments to make it easier to understand.

```
//Guess the secret number
#include <iostream>
using namespace std;

int main( )
{
   int secret, guess, absError, bestGuess, smallestError;

   //Generate the secret number
   srand(time(0));
   secret = 1 + rand( ) % 100;

   //Input the first guess
   cout << " Enter the first guess: ";
   cin >> guess;

   //Initialise smallest error and best guess
   smallestError = 100;
   bestGuess = -1;

   //Repeatedly input guesses until 0 is entered
   while (guess != 0)
   {
      //Keep track of the best guess and smallest error so far
      absError = abs(secret - guess);
      if (absError < smallestError)
      {
         smallestError = absError;
         bestGuess = guess;
         cout << "This is the best guess so far" << endl;
      }

      //Input the next guess
      cout << "Enter the next guess: ";
      cin >> guess;
   }

   //Display the best guess
   cout << "The secret number was " << secret << endl;
   cout << "The best guess was " << bestGuess << endl;
```

```
   return 0;
}
```

This was quite a complicated program to write, so if you did not get it right, do not worry. Just go through the program carefully, making sure you understand what each line does.

We have given a very basic version of this program, and you may want to extend it in several ways, such as displaying the purpose of the program and checking the input values for errors.

Important points in this lesson

Programming concepts

We have seen that a function is called by using its name with a list of parameters in parentheses. If there are no parameters (as in the `rand` function) then the parentheses are left empty. If the function takes a parameter (as in `time` and `abs`) then a value must be provided in the form of a variable or an expression. When functions (like `time`, `rand` and `abs`) return a value, we must do something with the value. There are many possibilities:

- We can call a function in an assignment statement:

  ```
  VariableName = FunctionName(Parameter);
  ```

  (Of course, *VariableName* must have been declared as a variable already.)

- The value returned by the function can be used in a more complicated expression, for example

  ```
  VariableName = 20 * FunctionName(Parameter) - 41;
  ```

  as we did with the `rand` function.

- We can also use a function call to provide a value to be output:

  ```
  cout << FunctionName(Parameter) << endl;
  ```

The `abs` function returns the absolute value of its integer parameter.

The `rand` function returns a large random integer value each time it is called. However, the first value it returns is always the same, and the values after that follow the same sequence unless the random number generator is seeded with the `srand` function.

The `time` function returns a large integer representing the value of the internal clock.

Apart from `abs`, `rand` and `time`, C++ has many other standard library (predefined) functions. A selection of these functions is given in Appendix D. You might find it interesting to take a look at them now.

Programming principles

Although we can use a function call to provide a value to be output as explained above, in general we prefer to introduce an additional variable and assign the value returned by the function to the variable before outputting its value.

To generate random numbers in a program, we use the `srand`, `time` and `rand` functions. We firstly use the `srand` and `time` functions to seed the random number generator with the statement:

```
srand(time(0));
```

In this statement, the `time` function returns an integer value (depending on the date and time of day) which is passed to the `srand` function. This ensures that the `rand` function (which should only be called after this statement) will not return the same random number each time the program is executed. The above statement only needs to be executed once (somewhere near the beginning of the program) even if the program needs to generate many random numbers with the `rand` function.

Random numbers are then generally created with a statement like:

```
RandomInteger = Smallest + Interval * (rand( ) % Many);
```

where

- *Many* is a literal value or variable representing how many possible values (i.e. the range of values that) are required,
- *Smallest* is a literal value or variable representing the smallest value in the required range, and
- *Interval* is the gap between values in the range.

Say for example you want to generate random even numbers between 20 and 100. Firstly, count how many possible values you want (there are 41 - check this), the smallest value is 20 and the interval is 2. Then the statement

```
randomEven = 2 * (rand( ) % 41) + 20;
```

will give a random value that complies with these restrictions.

Exercises

Exercise 17.1

Write a program to generate 6 random numbers for the South African Lottery. The numbers should all be between 1 and 49. Your program should reject duplicates by regenerating any numbers that have occurred already.

Exercise 17.2

What results will the following program fragments give? (Appendix D gives descriptions of a number of other standard library functions. These questions also involve operator precedence, so it may help to adapt the techniques we used in Lesson 2 to evaluate these expressions.)

(i)
```
float x = -6.32;
float y = fabs(x) * 2;
```
What is the resulting value of `y`?

(ii)
```
float x = 2;
float y = 2 * pow(x, 2) - 1;
float z = 2 * pow(x - 1, 2);
```
What are the resulting values of `y` and `z`?

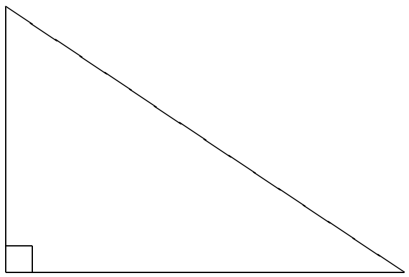
(iii)
```
float x = -4;
float y = sqrt(fabs(x));
```
What is the resulting value of `y`?

(iv)
```
float x = 3;
float y = 4;
float z = sqrt(pow(x, 2) + pow(y, 2));
```
What is the resulting value of `z`?

Exercise 17.3

About 2500 years ago there was a Greek guy called Pythagoras who was interested in politics, science and philosophy. He and his followers made many amazing discoveries.

One very important discovery of the Pythagoreans was a special characteristic of any triangle where one side is at right angles (i.e. perpendicular) to another side. If you take the length of each side of the triangle and multiply it by itself (i.e. square it), you will find that the squares of the two shorter sides always add up to the square of the longest side.



If we call the length of the longest side a, and the length of the other two sides b and c, we can express this as follows:

$$a = \sqrt{b^2 + c^2}$$

Write a program to ask for the lengths of the two shorter sides and then, using the `pow` and `sqrt` functions, to calculate and display the length of the longest side.

Exercise 17.4

There are two functions called `toupper` and `tolower` that can be used to convert a lowercase character to its uppercase equivalent and vice versa. Have a look at their definitions in Appendix D, and then write a short program to test the `toupper` function. It should input a character and display the result of applying function `toupper` to it. After running the program, you should be able to answer the following questions: What happens when `toupper` is applied to a lower case letter of the alphabet; what happens when it is applied to an upper case letter of the alphabet; what happens when it is applied to a non-alphabetic character (i.e. a digit or punctuation character)?

Exercise 17.5

The Infinite Monkey Theorem states that "If you have enough monkeys banging randomly on typewriters, they will eventually type the works of William Shakespeare."

Write a program to simulate such an experiment. The program should generate random letters (for simplicity's sake, generate uppercase letters only) to form words of random length. The shortest word should be 1

letter long (e.g. `I` or `A`) and the longest word should be 27 letters long (e.g. `HONORIFICABILITUDINITATIBUS` from *Love's Labour's Lost*).

Instead of getting the program to try to generate an entire play of Shakespeare, limit it to a single sentence of 15 words. Run the program a few times to see if you ever get any sensible words or phrases!

Lesson 18

Writing functions

Purpose of this lesson

In the activity and exercises of the previous lesson, we saw how to use standard library functions and became familiar with calling such functions. We can also define our own functions. In fact, we've been defining our own function from our first program, since the whole program is a function called `main`! In this lesson, we will learn to write more functions of our own.

Activity 18.a

Write a program that inputs three (floating point) numbers from the keyboard and outputs the maximum, i.e. the largest of the three.

You must write a function to determine the maximum of three values, and call the function in your program.

Test yourself

You should be able to write this program yourself without using a function. It will just require a number of nested `if` statements.

The problem statement requires you to write and use a function, however. The subactivities below show how this can be done.

Subactivity 18.a.i

Consider the following version of the `abs` function. We call it `myAbs`. Test it by using it in the program given in the solution to Activity 17.a in the previous lesson in place of the standard library function `abs`.

```
int myAbs(int i)
{
   if (i < 0)
      i *= -1;
   return i;
}
```

You should insert this function definition just before the `main` function.

Subactivity solution

```
1  //Guess the secret number
2  #include <iostream>
3  using namespace std;
4
5  int myAbs(int i)
6  {
7     if (i < 0)
8        i *= -1;
9     return i;
10 }
11
12 int main( )
13 {
14    int secret, guess, absError;
15
16    srand(time(0));
17    secret = 1 + rand( ) % 100;
18
19    cout << "Enter the guess: ";
20    cin >> guess;
21
22    absError = myAbs(secret - guess);
23    cout << "You were out by " << absError << endl;
24
25    return 0;
26 }
```

Discussion

Note the layout of the function definition

```
ReturnType FunctionName(ParameterList)
{
   Statements;
   return ReturnValue;
}
```

ReturnType describes the type of value that will be returned by the function. In the case of `myAbs`, we want the function to return an integer, so the return type is `int`.

FunctionName is the name of the function that we want to call. The rules for what are legal names of functions are exactly the same as for variables. (Take a look back at Lesson 3 for the legal names for variables.)

ParameterList specifes the parameters that the function will take. If there are no parameters, the parentheses are left empty. Otherwise the type and the name of (each of) the parameter(s) is specified. If there is more than one parameter (the standard library function `pow` is an example of such a function) the parameters are separated by commas. A parameter list looks similar to the declaration of one or more variables.

Statements represents any number of valid C++ statements. (Note that we always use braces to specify the body of a function, even if there is only one statement in its body.)

The `return` statement specifies the value that is returned by the function. *ReturnValue* can be a variable or an expression, but its type must be the same as *ReturnType* specified in the function header. For example, if *ReturnType* is `float`, then *ReturnValue* must represent a floating point number.

Finally, note the function call `myAbs(secret - guess)`. We use the same name as the name in the function definition, and provide a value for the parameter in parentheses.

We've been talking about the parameter in the function call, and the parameter in the function definition, which causes some confusion. We distinguish between the two as follows: The value provided in the function call is called the *actual parameter* and the name used in the parameter list of the function definition is called the *formal parameter*. So in the above program, the value of `secret - guess` represents the actual parameter, and `i` represents the formal parameter.

To see how things work, consider a simpler situation where we call function `myAbs` with a literal value of -3, for example

```
cout << myAbs(-3) << endl;
```

The first thing that happens is that the call of `myAbs` is executed. (The computer must do this before trying to output anything, because it must first determine what to output.) Actual parameter -3 is copied to formal parameter `i` in the function definition and the statements of function `myAbs` are executed one by one. We test whether `i < 0` (yes it is) and so we multiply `i` by -1 (-3 times -1 is +3, so `i` gets the value 3). Finally we return 3 as the value of the function. The function terminates and we return to where `myAbs` was called. Now `cout` has a value to output, namely 3, which it does.

Things would work exactly the same for the slightly more complicated situation where the actual parameter is an expression, like `secret - guess`. In this case, the computer has to evaluate `secret - guess` first before `myAbs` can be called, to provide the value for formal parameter `i`.

Subactivity 18.a.ii

In pencil, fill in the missing values in the following series of variable diagrams (i.e. the boxes with nothing in them). We have filled in 23 for the number generated by line 17, and 51 for the number entered by the user in line 20.

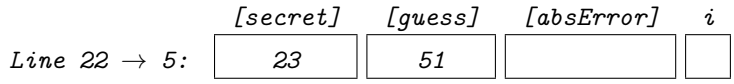
Line 17: `secret = 1 + rand( ) % 100;`

	secret	*guess*	*absError*
Line 17:	*23*	*?*	

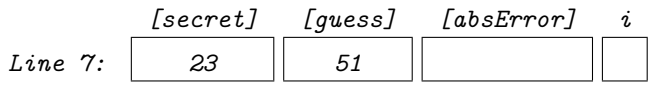
Line 20: `cin >> guess;`

	secret	*guess*	*absError*
Line 20:	*23*	*51*	

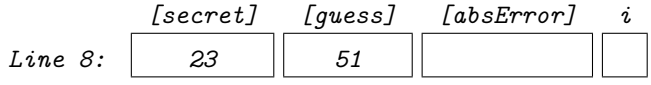
Line 22 → 5: `absError = myAbs(secret - guess);`



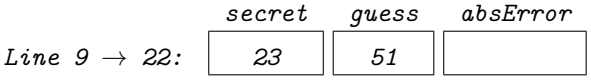
Line 7: `if (i < 0)`



Line 8: `i *= -1;`



Line 9 → 22: `return i;`

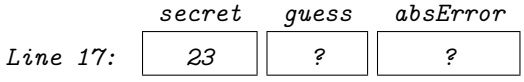


Before you fill in the variable diagrams, note the following things:

- The notation 22 → 5 indicates that program execution jumps from line 22 to line 5.
- We put the names of variables `secret`, `guess` and `absError` in square brackets for lines 5, 7 and 8 because although these variables still exist, they are inaccessible to function `myAbs`.

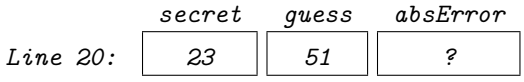
Subactivity solution

Line 17: `secret = 1 + rand( ) % 100;`



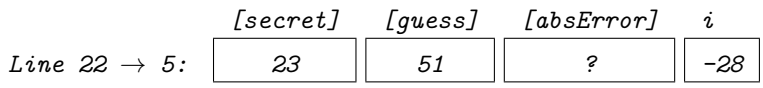
Like `guess`, `absError` is uninitialised at this stage of the program, and so we put a `?` in its box to show that its value is unpredictable.

Line 20: `cin >> guess;`



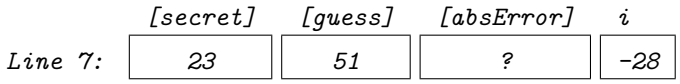
The user enters 51 at the keyboard, and this value is stored in variable `guess`.

Line 22 → 5: `absError = myAbs(secret - guess);`



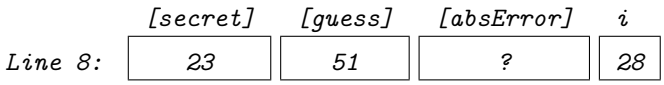
Before function `myAbs` starts executing, the expression `secret - guess` is calculated and this value (-28) is sent as the actual parameter. When `myAbs` starts executing, formal parameter `i` is created and the value sent as the actual parameter is stored in it. The three local variables `secret`, `guess` and `absError` are hidden from function `myAbs` because they are declared in the `main` function.

Line 7: `if (i < 0)`



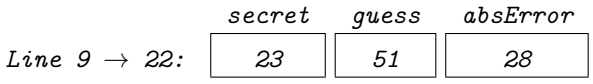
This statement has no effect on any of the variables. It simply tests whether the value of `i` is negative. Since `i`'s value is negative, line 8 is executed:

Line 8: `i *= -1;`



The value of `i` is multiplied by -1 and stored back in `i`.

Line 9 → 22: `return i;`



The `return` statement terminates the function and returns the value of the expression after it (in this case `i`, and whose value is 28) as the value of the function. Formal parameter `i` is destroyed. As soon as program execution returns to the calling function, its local variables become visible again. The assignment statement in line 22 causes the value of the function (28) to be stored in local variable `absError`.

Discussion

We stated above that local variables of the calling function are inaccessible (or hidden) from the called function. You might like to see what happens if you attempt to refer to variables `guess` or `secret` in function `myAbs`.

You will see that the compiler complains (with an error message), saying that the particular variables are undeclared.

Subactivity 18.a.iii

Write an alternative version of `myAbs` function that contains two `return` statements: one to return `-i`, and the other to return the unchanged `i`.

Subactivity solution

There are (at least) two ways of doing this:

```
int myAbs(int i)
{
   if (i < 0)
      return -i;
   else
      return i;
}
```

and

```
int myAbs(int i)
{
   if (i < 0)
      return -i;
   return i;
}
```

We prefer the first version because it shows clearly what is done in the two possibilities. Although the second version is less readable (it is less clear what value is actually going to be returned) it is interesting because it illustrates an important property of the `return` statement, namely it causes the function to terminate immediately. Normally, the statement after an `if` statement (and we don't mean the `else` part) is executed no matter whether the condition of the `if` statement is `true` or `false`. In this case however, if `i` is negative the `return` statement inside the `if` is executed and the program jumps out of the function immediately without executing the rest of the statements in `myAbs`. On the other hand, if `i` is non-negative, the first `return` statement will be skipped and the final `return` statement will be executed.

Test both of the above versions of `myAbs` in your program to convince yourself that they do work. Use appropriate positive, zero and negative values to test both versions.

Subactivity 18.a.iv

In your study notebook, write a function called `max2` that determines the maximum of two floating point numbers. Hint: First write the function header. The function should take two floating point values as parameters (call them `x` and `y`) and return a floating point value.

Subactivity solution

The function header should be

```
float max2(float x, float y)
```

Note that you can't use the shorthand notation that we use for variable declarations and write

```
float max2(float x, y)
```

This is not allowed.

The body of the function should simply compare `x` and `y` and return the one which is greater. In other words

```
float max2(float x, float y)
{
   if (x > y)
      return x;
   else
      return y;
}
```

We could get by without the `else` (if you don't know why, turn back to the solution to the previous subactivity) but once again we prefer this solution.

Subactivity 18.a.v

Now see if you can write a function called `max3` that returns the greatest of three floating point values.

One way would be to use nested `if` statements. However, if you try this, you'll see that it gets quite messy. A much simpler way does not contain any `if` statements at all! It simply calls `max2` twice.

Subactivity solution

We first give a way of writing the function using nested `if` statements:

```
float max3(float x, float y, float z)
{
   if (x > y)
      if (x > z)
         return x;
      else
         return z;
   else
      if (y > z)
         return y;
      else
         return z;
}
```

This is long and complicated, and we had to test it with all possible combinations of values for `x`, `y` and `z` to be sure that it worked correctly.

Another solution uses consecutive (rather than nested) `if` statements with compound conditions:

```
float max3(float x, float y, float z)
{
   if (x >= y && x >= z)
      return x;
   if (y >= x && y >= z)
      return y;
   return z;
}
```

Note two things about this solution: Firstly, we use `>=` instead of `>` because the function wouldn't work with `>`. (Using `>`, try tracing the function when `x` is equal to `y`, and they are both greater than `z`.) Secondly, this solution uses the trick explained above, namely we assume that `return z;` will only be executed if neither `return x;` or `return y;` has been executed.

An even simpler way is to introduce a local variable:

```
float max3(float x, float y, float z)
{
   float max = x;
   if (y > max)
      max = y;
   if (z > max)
      max = z;
   return max;
}
```

Variable `max` is initialised to the value of `x`, and then updated each time a better candidate is found.

The final version that calls `max2` is the simplest of all:

```
float max3(float x, float y, float z)
{
   return max2(x, max2(y, z));
}
```

We think you'll agree that this is the most elegant solution. It is interesting for two reasons: (i) `max3` is a function that calls another function, namely `max2`, and (ii) the single `return` statement uses a complicated expression to determine the value to be returned. The expression is evaluated first (by calling `max2` twice) before the value can be returned.

Subactivity 18.a.vi

Write a program that uses the `max3` function to fulfil the requirements of the main activity. (Page back to Activity 18.a if you can't remember what it was supposed to do.)

Activity solution

One solution is:

```
1  //Determine the maximum of three values
2  #include <iostream>
3  using namespace std;
4
5  float max2(float x, float y)
6  {
7     if (x > y)
8        return x;
9     else
10       return y;
```

```
11 }
12
13 float max3(float x, float y, float z)
14 {
15    return max2(x, max2(y, z));
16 }
17
18 int main( )
19 {
20    float a, b, c, largest;
21    cout << "Enter three numbers: ";
22    cin >> a >> b >> c;
23    largest = max3(a, b, c);
24    cout << "The maximum is: " << largest << endl;
25    return 0;
26 }
```

Short and sweet!

Discussion

Take a look at functions `max2` and `max3` again. They have parameters with names in common. Although the functions will work perfectly well with these parameter names, it causes confusion. Think about it: for the inner call of `max2` (the one that would have to be done first) we send actual parameter `y` to formal parameter `x` and actual parameter `z` to formal parameter `y`.

This arrangement works because the actual and formal parameters are completely separate. Separate memory space (like for a variable) is set aside for each formal parameter, and the value of the actual parameter is copied to it. As we saw in Subactivity 18.a.ii where you had to fill in the trace table for parameter passing, all variables in the calling function, in this case `max3` (whether they are actual parameters or not), are inaccessible in the called function, in this case `max2`. So `x` and `y` in `max2` are completely independent of `x`, `y` and `z` in `max3`. One of the exercises at the end of this lesson requires you to draw variable diagrams for this program.

I am sure you would agree that this can only cause confusion. We therefore introduce a convention that we will try to maintain throughout the rest of this study guide: *Always use different names for the parameters of a function from other variables in the program.*

Activity 18.b

Write a program that calculates the compound interest on an amount of money invested at two different banks, at different interest rates. For example, Bank A gives an interest rate of 9% but requires the amount to be invested for 5 years. Bank B gives an interest rate of 10% but requires the amount to be invested for 8 years.

A typical execution of the program should look like this:

```
Enter the amount to be invested: 1000
======
Bank A
```

```
======
Enter the interest rate: 9
How many years will the amount be invested? 5
======
Bank B
======
Enter the interest rate: 10
How many years will the amount be invested? 8
======
After 5 years, R1000 invested at 9% will yield R 1538.62
After 8 years, R1000 invested at 10% will yield R 2143.59
```

Note that your program need not check that the number of years is greater than the required limit. This is to allow the program to be used for comparisons of other banks that have different requirements.

Test yourself

This problem is perhaps bigger than those we have seen so far. We are going to follow a strategy of tackling a small part of the problem, but in such a way that we can use everything we do in the final solution.

The subactivities below show one way of tackling this problem step-by-step.

Instead of writing out an algorithm for the entire program (as we have done previously) we are first going to tackle a smaller, simpler version of the program, namely to make a calculation for one bank. Then we can duplicate the code to deal with two banks.

Subactivity 18.b.i

Activity 15.a showed how to calculate the compound interest on an amount over a number of years. We could use two for loops to calculate the two totals (as in the solution to Activity 15.a) but there is a simpler way. There is a single formula that we can use. Say *Amount* is the investment amount, *Rate* is the interest rate and $Term$ is the number of years, then the formula

$$Total = Amount * (1 + Rate/100)^{Term}$$

will give the final value of the investment after $Term$ years.

Write a function to do this calculation. It should take three parameters (two floating point values and one integer) and return a (floating point) result.

Subactivity solution

```
float compoundInterest(float amountP, float rateP, int termP)
{
   return amountP * pow(1+rateP/100, termP);
}
```

We use the letter `P` at the end of the formal parameter names to distinguish them from the actual parameter names.

Note that this function does not require any `if` statements or loops. It contains an expression in a single `return` statement. Many functions that you will write will look like this.

Subactivity 18.b.ii

Now write a program that inputs the data for a single bank, uses the above function to calculate the final value of the investment and displays it.

Subactivity solution

```
//Calculates compound interest on an investment amount
#include <iostream>
#include <cmath>
using namespace std;

float compoundInterest(float amountP, float rateP, int termP)
{
   return amountP * pow(1 + rateP/100, termP);
}

int main( )
{
   float amount, rate, total;
   int term;

   cout << "Enter the amount to be invested: ";
   cin >> amount;
   cout << "Enter the interest rate: ";
   cin >> rate;
   cout << "How many years will the amount be invested? ";
   cin >> term;

   total = compoundInterest(amount, rate, term);

   cout.setf(ios::fixed);
   cout.precision(2);

   cout << "After " << term << " years, R" << amount;
   cout << " invested at " << rate << "% will yield R";
   cout << total << endl;

   return 0;
}
```

Subactivity 18.b.iii

Now you should be able to write the entire program. It will require some copying and pasting of code, and a few additional `cout` statements.

Activity solution

```
//Compares compound interest on investments at two banks
#include <iostream>
#include <cmath>
using namespace std;

float compoundInterest(float amountP, float rateP, int termP)
{
   return amountP * pow(1 + rateP/100, termP);
}

int main( )
{
   float amount, rateA, totalA, rateB, totalB;
   int termA, termB;

   cout << "Enter the amount to be invested: ";
   cin >> amount;

   cout << "======" << endl << "Bank A";
   cout << endl << "======" << endl;
   cout << "Enter the interest rate: ";
   cin >> rateA;
   cout << "How many years will the amount be invested? ";
   cin >> termA;

   cout << "======" << endl << "Bank B";
   cout << endl << "======" << endl;
   cout << "Enter the interest rate: ";
   cin >> rateB;
   cout << "How many years will the amount be invested? ";
   cin >> termB;

   totalA = compoundInterest(amount, rateA, termA);
   totalB = compoundInterest(amount, rateB, termB);

   cout.setf(ios::fixed);
   cout.precision(2);

   cout << "After " << termA << " years, R" << amount;
   cout << " invested at " << rateA << "% will yield R";
   cout << totalA << endl;

   cout << "After " << termB << " years, R" << amount;
   cout << " invested at " << rateB << "% will yield R";
   cout << totalB << endl;

   return 0;
}
```

Discussion

This program illustrates another reason for making sure that the names of formal parameters are different to all other variables in the program. It allows us to call the same function more than once with different actual parameters. Although one might argue that no confusion would be caused in the program given in the solution to Subactivity 18.b.ii if the same names were used for the formal and actual parameters, in our final program it would be misleading to have the names match for the first call but not for the second.

Important points in this lesson

Programming concepts

The general layout of a function is

```
ReturnType FunctionName(ParameterList)
{
   Statements;
}
```

Statements should contain (at least one) `return` statement that returns a value matching *ReturnType*.

Note how the `main` functions that we have been writing have all complied with this structure. The return type is `int`, the parameter list is empty, and the program consists of C++ statements including a `return` statement that returns a value matching the return type of `main` (i.e. `int`).

The format of a `return` statement is

```
return ReturnValue;
```

where *ReturnValue* can be an expression with the same type as the *ReturnType* of the function.

When a `return` statement is executed, the function terminates immediately, and any other statements in the body of the function are skipped.

Apart from these features, functions must be used in the following ways:

- When calling a function, the parameters must correspond: There must be the same number of actual parameters in the calling statement as there are formal parameters in the subprogram header, and their types must match.
- A function is called by using its name in an expression, much as one uses a variable name except that the name must always be followed by parentheses containing the actual parameters (if any).
- Each function is an independent subprogram unit. This means that local variables of a calling function can only be accessed by that function itself and not by the functions that it calls. If such functions need the values of local variables, they must be sent as parameters.

Programming principles

Like variables, it's good to choose descriptive names for functions. The name of a function should describe the value that is returned, since the function call will generally be used to provide a value in an expression.

Also, to avoid confusion between the names of actual and formal parameters, we follow the convention of using different names for the formal parameters of a function from all other variables in a program, especially the actual parameters.

Exercises

Exercise 18.1

Look at the following program skeleton:

```
1  // Illustrates reference parameters
2  #include <iostream>
3  using namespace std;
4
5  bool test(float par1, float par2, float par3)
6  {
7
8  }
9
10 int main( )
11 {
12    float var1, var2, var3;
13
14    if (test(var1, var2, var3))
15    {
16
17    }
18
19    return 0;
20 }
```

Now, in the table below, fill in the line numbers opposite the appropriate concepts:

Concept	*Line no(s)*
Function heading	
Formal parameters	
Return type	
Function call	
Actual parameters	
First line executed	

Exercise 18.2

For every triangle, the sum of (the lengths of) any two sides is greater than the third side. This fact can be used to test whether three numbers represent the valid sides of a triangle.

Using the program skeleton above, write a program that inputs three numbers and calls a function to check whether they represent the sides of a triangle. Hint: You'll find it useful to write an additional boolean function to test whether the sum of two values is greater than a third.

Exercise 18.3

There is a cricket (an insect) that chirps faster as it gets hotter. If one counts the number of chirps this insect makes in one minute, adds 160 to that number and then divides the answer by 4, one gets the approximate temperature in degrees Fahrenheit. In other words:

$$fahrenheitTemperature = (chirpsPerMinute + 160)/4$$

Write a program, using functions as appropriate, that reads in the number of chirps per minute and then gives the approximate temperature in both Fahrenheit and Celsius.

The formula for converting Fahrenheit to Celsius is:

$$c = (f - 32) * 5/9$$

Exercise 18.4

One way to calculate the area of a triangle is to measure each side (call them a, b and c) and then to use the following formulas:

$$s = (a + b + c)/2$$

and

$$area = \sqrt{s * (s - a) * (s - b) * (s - c)}$$

Write a program that reads in the three sides of a triangle and then displays the area. Use the fact that a triangle with sides of 3, 4, and 5 has an area of 6, and that a triangle of sides 5, 12 and 13 has an area of 30 to test your program.

Exercise 18.5

Draw variable diagrams for the program given in the solution to Activity 18.a (with the functions `max2` and `max3`). Assume that the user enters the values 4, 5 and 3 for variables `a`, `b` and `c` respectively.

Lesson 19

Local and global variables

Purpose of this lesson

We can declare variables within our own functions. These are called local variables. Like the formal parameters of a function, the lifetime of a local variable is from when the function starts executing until it terminates.

Global variables are variables that are not declared in a function, but are accessible within it. In general, global variables are a bad idea - we will see why.

Activity 19.a

The following program contains a function called `iPow` that calculates `x` to the power of `n`, where `n` is a non-negative integer. (The standard library function `pow` does not require the exponent to be an integer. See Appendix D.)

```
1  //Raises one number to an integer power
2  #include <iostream>
3  using namespace std;
4
5  float x, answer;
6  int i, n;
7
8  float iPow( )
9  {
10    answer = 1;
11    for (i = 1; i <= n; i++)
12       answer *= x;
13    return answer;
14 }
15
16 int main( )
17 {
18    cout << "Enter a number and a non-negative integer: ";
19    cin >> x >> n;
20    cout << x << " to the power " << n << " = ";
21    cout << iPow( ) << endl;
22    return 0;
23 }
```

Note that this program uses a trick. By declaring all variables before functions `iPow` and `main`, all functions have access to them. This dispenses with the need for parameter passing.

Change the program so that it inputs two pairs of numbers and integers, calculates the powers, and displays the sum.

The output of a test run of the program should look like this:

```
Enter a number and a non-negative integer: 2 4
Enter a number and a non-negative integer: 3 2
The sum of
2 to the power 4 and
3 to the power 2 is
25
```

Test yourself

Obviously, we need to call the `iPow` function twice.

If you take a moment's thought about it, you should agree that we will have a problem simply calling the function an additional time. If we input the two pairs of numbers into the same variables, the original values will be overwritten. So we need some additional variables. But then to ensure that the function works with the correct values, we'll have to copy values to the correct variables before calling the function each time.

This will all become very messy. The point of the activity is to convince you that global variables as used in the above program are not a good idea. We should rather use local variables and parameters.

If you managed to change the program by using parameters and local variables, good! Compare your answer with ours given in the activity solution. If you managed to do it by using more global variables, bad! Take a look at Subactivity 19.a.ii to see why this is a bad idea. Then try to fix your program so that it doesn't use any global variables.

The subactivities below take you step-by-step through the changes that need to be made, and also show why it is not good to use global variables.

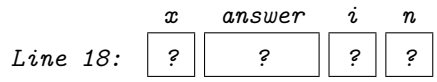
Before you attempt the main activity, you should make sure that the above program actually works. The first subactivity gets you to test it.

Subactivity 19.a.i

In your study notebook, draw a series of variable diagrams for the given program. Assume that the user enters 5 and 3 for the input and hence predict what the output will be. Start your variable diagrams where the program starts executing in line 18.

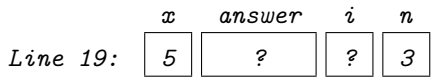
Subactivity solution

Line 18: `cout << "Enter a number and a non-negative integer:  ";`



When the program starts executing, all four global variables exist already (and will remain in existence until the program terminates). They are all uninitialised at this stage.

Line 19: `cin >> x >> n;`



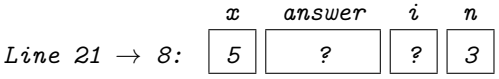
We are told to assume that the user enters 5 and 3 when prompted.

Line 20: `cout << x << " to the power of " << n < " = ";`

This line doesn't change the value of any variables, so we don't repeat the variable diagrams.

Line 21 → 8: `cout << iPow( ) << endl;`

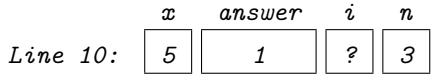
Before `cout` can output a value, the function call `iPow( )` must be evaluated.



Since `x`, `answer`. `i` and `n` are global variables, they are not hidden in `iPow` and so we do not put square brackets around their names.

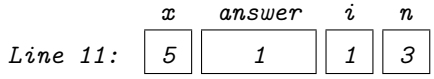
Since `iPow` has no parameters, no additional variable boxes are drawn.

Line 10: `answer = 1;`



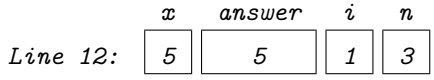
Straightforward!

Line 11: `for (i = 1; i <= n; i++)`



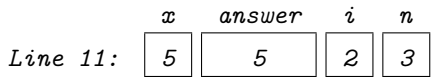
Variable `i` is initialised to 1. It is compared with the value of `n`, and since `i` is less than or equal to `n`, the body of the loop is executed.

Line 12: `answer *= x;`



The value of `answer` (1) is multiplied by the value of `x` (5), and the result is stored back in `answer`.

Line 11: `for (i = 1; i <= n; i++)`



Variable `i` is incremented by 1. It is compared with the value of `n`, and since `i` is less than or equal to `n`, the body of the loop is executed.

Line 12: `answer *= x;`

	x	*answer*	*i*	*n*
Line 12:	*5*	*25*	*2*	*3*

The value of `answer` (5) is multiplied by the value of `x` (5), and the result is stored back in `answer`.

Line 11: `for (i = 1; i <= n; i++)`

	x	*answer*	*i*	*n*
Line 11:	*5*	*25*	*3*	*3*

Variable `i` is incremented by 1. It is compared with the value of `n`, and since `i` is less than or equal to `n`, the body of the loop is executed.

Line 12: `answer *= x;`

	x	*answer*	*i*	*n*
Line 12:	*5*	*125*	*3*	*3*

The value of `answer` (25) is multiplied by the value of `x` (5), and the result is stored back in `answer`.

Line 11: `for (i = 1; i <= n; i++)`

	x	*answer*	*i*	*n*
Line 11:	*5*	*125*	*4*	*3*

Variable `i` is incremented by 1. It is compared with the value of `n`, and since `i` is not less than or equal to `n`, the body of the loop is not executed.

Line 13 → 21: `return answer;`

This statement doesn't change the values of any variables. It returns the value of `answer` as the value of the function. Now the `cout` statement in line 18 can do its work, namely to output the value of `answer` on the screen.

Discussion

Note the following: Since variables `x`, `answer`, `n` and `i` are declared before functions `iPow` and `main`, and are in the global namespace, i.e. they are accessible to both functions and both functions can refer to them and change their values. We therefore do not put brackets round their names in the variable diagrams at any stage of the program.

Subactivity 19.a.ii

The program below is meant to calculate and display all the powers of 2 less than some number, obtained from the user. For example, say the user enters 50, then the program should display

```
2 to the power 1 is 2
2 to the power 2 is 4
2 to the power 3 is 8
2 to the power 4 is 16
2 to the power 5 is 32
```

Here is the program:

```
1  //Displays all powers of 2 less than a limit
2  #include <iostream>
3  using namespace std;
4
5  float x, answer;
6  int i, n;
7
8  float iPow( )
9  {
10    answer = 1;
11    for (i = 1; i <= n; i++)
12       answer *= x;
13    return answer;
14 }
15
16 int main( )
17 {
18    cout << "Enter a limit for the calculations: ";
19    cin >> answer;
20    x = 2;
21    n = 1;
22    while (iPow( ) < answer)
23    {
24       cout << x << " to the power " << n << " = ";
25       cout << iPow( ) << endl;
26       n++;
27    }
28    return 0;
29 }
```

When the program is executed, however, it inputs the value but then does nothing. It simply terminates. Show, by means of a series of variable diagrams, why it doesn't work. Assume that the user enters the number 1000.

Subactivity solution

Line 18: `cout << "Enter a limit for the calculations:  ";`

	x	answer	i	n
Line 18:	?	?	?	?

Line 19: `cin >> answer;`

	x	answer	i	n
Line 19:	?	1000	?	?

Line 20: `x = 2;`

	x	answer	i	n
Line 20:	2	1000	?	?

Line 21: `n = 1;`

	x	answer	i	n
Line 21:	2	1000	?	1

Line 22 → 8: `while (iPow( ) < answer)`

This line doesn't change the value of any variables, and it doesn't create or rename any parameters, so we don't repeat the variable diagrams.

Line 10: `answer = 1;`

	x	answer	i	n
Line 10:	2	1	?	1

Line 11: `for (i = 1; i <= n; i++)`

	x	answer	i	n
Line 11:	2	1	1	1

Variable `i` is initialised to 1. It is compared with the value of `n`, and since `i` is less than or equal to `n`, the body of the loop is executed.

Line 12: `answer *= x;`

	x	answer	i	n
Line 12:	2	2	1	1

The value of `answer` (1) is multiplied by the value of `x` (2), and the result is stored back in `answer`.

Line 11: `for (i = 1; i <= n; i++)`

	x	*answer*	*i*	*n*
Line 11:	*2*	*2*	*2*	*1*

Variable `i` is incremented by 1. It is compared with the value of `n`, and since `i` is not less than or equal to `n`, the body of the loop is not executed.

Line 13 → 22: `return answer;`

The value of `answer` (2) is returned as the value of the function. This is compared with the current value of `answer` (also 2) and since 2 is not less than 2, the condition fails and the body of the `while` loop is never executed.

Discussion

The problem is simply that function `main` and function `iPow` are both using variable `answer` for their own purposes and consequently messing up what the other is trying to do.

In this example, it is quite easy to see that this is the problem. The obvious solution is to introduce a separate variable for each function to use. For example, we could introduce a variable called `limit` for the main function to use.

The point is that in the above program it is not clear which function variable `answer` belongs to. In a big program, using global variables can be particularly dangerous because there can be many, many variables, and if it is not clear which variable belongs to which function, some function can innocently change the value of a variable without knowing the effect this might have on some other function that also uses it.

The moral of the story is that we should rather declare variables locally to show where they belong. This not only makes a program more readable (we can see exactly what's going on) it also prevents other functions from messing around with those variables. Remember that a locally declared variable is inaccessible to other functions.

Subactivity 19.a.iii

Now explain (i.e. write in your study notebook) how you would change the program in the main activity to make variables `answer` and `i` local rather than global variables.

Subactivity solution

- *Remove the declarations of answer and i from the beginning of the program.*
- *Declare and initialise answer at the beginning of function iPow:*

  ```
  float answer = 1;
  ```

- *Declare variable i in the for loop:*

  ```
  for (int i = 1; ...etc.
  ```

Subactivity 19.a.iv

The next question we need to ask is about variables `x` and `n`: Who do they belong to? The program depends on the fact that `x` and `n` are used by both the `main` function and function `iPow`.

In this case, we can argue as follows: Both `x` and `n` get their initial values from function `main`. Then `main` calls `iPow` which just uses their values to make a calculation. These facts tell us that `x` and `n` belong to `main` rather than `iPow`.

Once you've decided which variables belong to which function, you need to make them local variables of that function. Then if another function needs their values, they should be sent as parameters rather than making them global variables of both functions.

Change function `iPow` so that it has two formal parameters: `xP` (to receive a floating-point value) and `nP` (to receive an integer).

Also incorporate the changes specified in Subactivity 19.a.iii.

Subactivity solution

```
float iPow(float xP, int nP)
{
   float answer = 1;
   for (int i = 1; i <= nP; i++)
      answer *= xP;
   return answer;
}
```

Note that `answer` and `i` are now local variables. Also make sure that you have changed `x` and `n` to `xP` and `nP` in the body of the function as we have done above.

Discussion

This subactivity introduced an important reason for using parameters, namely to make the sharing of variables explicit.

When it has a formal parameter list, a function says to the world "The only information I need to do my work are these values". Parameters make a program easier to read since they make it clear what values each function requires to do its work.

Subactivity 19.a.v

Fix the `main` function of the original program to call function `iPow` using these parameters: Move the declaration of `x` and `n` to the `main` function (i.e. make them local variables of `main`) and include them as actual parameters in the call of `iPow`.

Subactivity solution

```
//Raises one number to an integer power
```

```
#include <iostream>
using namespace std;

float iPow(float xP, int nP)
{

   float answer = 1;
   for (int i = 1; i <= nP; i++)
      answer *= xP;
   return answer;
}

int main( )
{
   float x;
   int n;
   cout << "Enter a number and a non-negative integer: ";
   cin >> x >> n;
   cout << x << " to the power " << n << " = ";
   cout << iPow(x, n) << endl;
   return 0;
}
```

Subactivity 19.a.vi

Now you are ready to tackle the main activity. (If you can't remember what it requires, flip back to it briefly.)

You will need to declare additional variables (for the second pair of numbers). It would also be a good idea to declare a variable to hold the sum since we were naughty to call a function in an output statement. The assignment statement for sum should simply call `iPow` twice (with the respective pairs of numbers) and add them together.

Do it now. Test your program with some simple values to see whether it works correctly.

Activity solution

```
//Raises two numbers to integer powers and adds them
#include <iostream>
using namespace std;

float iPow(float xP, int nP)
{
   float answer = 1;
   for (int i = 1; i <= nP; i++)
      answer *= xP;
   return answer;
}

int main( )
{
   float x1, x2;
```

```
   int n1, n2;
   float sum;

   cout << "Enter a number and a non-negative integer: ";
   cin >> x1 >> n1;

   cout << "Enter a number and a non-negative integer: ";
   cin >> x2 >> n2;

   sum = iPow(x1, n1) + iPow(x2, n2);

   cout << "The sum of" << endl;
   cout << x1 << " to the power " << n1 << " and " << endl;
   cout << x2 << " to the power " << n2 << " is ";
   cout << sum << endl;

   return 0;
}
```

Important points in this lesson

Programming concepts

A global variable is a variable that is declared before all function definitions, and can be accessed by all functions in the program. A local variable is declared in the body of a function, and can only be accessed by that function.

Programming principles

In this lesson we have seen some important things about functions and parameters:

- We should use local rather than global variables when variables belong to specific functions. This prevents functions from inadvertently changing the values of variables needed in other functions.
- We should use parameters rather than global variables when more than one function needs access to the same variables. This makes it clear which variables need to be shared or which values need to be transferred from one function to another, and hence increases the readability and reliability of programs.

If the above rules are followed, the formal parameter list serves as a clear indication of what values a function requires to do its work. Even though it may use local variables to do its work, this is no concern of the rest of the program. This makes a program more readable, and allows a function to be re-used in other programs.

Exercises

Exercise 19.1

Fix the program of Subactivity 19.a.ii so that it uses only local variables and parameters (and no global variables) to do what it was intended to do.

Exercise 19.2

Study the following program (consisting of three functions) and state in terms of line numbers where (i.e. in which statements) each variable is accessible.

```
1  //Determines the sum and product of 1 to n
2  #include <iostream>
3  using namespace std;
4
5  int n, answer;
6
7  int sum( )
8  {
9     for (int i = 1; i <= n; i++)
10       answer += i;
11    return answer;
12 }
13
14 int product( )
15 {
16    int answer = 1;
17    for (int i = 2; i <= n; i++)
18       answer *= i;
19    return answer;
20 }
21
22 int main( )
23 {
24    answer = 0;
25    cout << "Enter a positive integer: ";
26    cin >> n;
27    cout << "The sum of 1 up to " << n << " is ";
28    cout << sum( ) << endl;
29    cout << "The product of 1 up " << n << " is ";
30    cout << product( ) << endl;
31    return 0;
32 }
```

Exercise 19.3

Rewrite the program of Exercise 19.2 above so that there are no global variables. Use local variables and parameters instead. Also introduce two new variables to fix the problem of calling the functions in output statements.

Lesson 20

Void functions

Purpose of this lesson

In Lessons 17 and 18 we looked at functions that return values, both standard library functions and our own functions. When we wrote our own functions, we saw that such functions should have a `return` statement which returns a value of the same type as the return type specified before the function name. We saw that the value returned by such a function is used by the calling statement. In other words, the function can be called in an expression, in an assignment statement or in an output statement (although we prefer not to do that).

But what about functions that don't return values? We have seen a few standard library functions like that. Think of the `getline` function (Lesson 7) and the `srand` function (Lesson 17). Both of these functions are used as stand-alone statements, e.g.

```
getline(cin, name, '\n');
```

and

```
srand(time(0));
```

These are called *void functions* (as opposed to *value-returning functions*). In this lesson we see how to write our own void functions.

Activity 20.a

Write a program to display a histogram (in the form of rows of characters) for values entered from the keyboard. In particular, the program must input all the values (only positive integers allowed - input is terminated by a negative number) and display a row of asterisks for each value. Then it must calculate the average of the values and display a row of plus signs for that.

The output of a test run should look like this:

```
Enter the values (negative to end):
20
********************
10
**********
1
*
9
*********
-1
The average is 10
++++++++++
```

Test yourself

You should realise straight away that we will need a number of loops for this program to deal with the repetition.

A more subtle idea is to be able to recognise when to use a function to make life easier for yourself.

For this program we are going to use a function to display one row of characters, and call it wherever needed.

If you managed to write a program to solve this problem on your own, well done! Compare your function with the one used in the activity solution.

If you are cautious and don't want to miss out on any of the explanations along the way, you can attempt the following subactivities.

Subactivity 20.a.i

In your study notebook, write a void function to display a row of `n` asterisks on the screen (where `n` is a parameter of the function).

This is not as difficult as you might imagine. Use the following framework for your function:

```
void displayRow (int n)
{
   :
}
```

The body of your function should consist of a `for` loop that runs from 1 to `n` and displays one asterisk for each repetition.

Subactivity solution

Here is our version:

```
void displayRow (int n)
{
   for (int i = 1; i <=n; i++)
      cout << '*';
   cout << endl;
}
```

Note two things about this function:

- It does not have a return type (only the reserved word `void`) and there is no `return` statement.
- We included an additional `cout` statement after the `for` loop to end the line (i.e. make the cursor jump to the next line).

Subactivity 20.a.ii

Now write a simple program to test this function. The program must input a positive integer and call the function to display a row of asterisks that long.

Subactivity solution

```
//Display a row of asterisks
#include <iostream>
using namespace std;

void displayRow(int n)
{
   for (int i = 1; i <= n; i++)
      cout << '*';

   cout << endl;
}

int main( )
{
   int m;

   cout << "Enter a positive integer: ";
   cin >> m;

   displayRow(m);

   return 0;
}
```

Note that according to our convention we use different names for the actual and formal parameters (even though the program would work perfectly well if we used the same names).

Also note how we call the `displayRow` function. Since it does not return a value, we don't have to call the function in an expression. We call it as a statement on its own.

Subactivity 20.a.iii

Adapt the above program to display a triangle of asterisks instead of a single row. In other words, if the user enters 5, the following should be output:

```
*
**
***
****
*****
```

Hint: Use a for loop in the `main` function and call the `displayRow` function in it.

Subactivity solution

```
//Display a triangle of asterisks
#include <iostream>
using namespace std;

void displayRow(int n)
```

```
{
   for (int i = 1; i <= n; i++)
      cout << '*';

   cout << endl;
}

int main( )
{
   int m;

   cout << "Enter a positive integer: ";
   cin >> m;

   for (int j = 1; j <= m; j++)
      displayRow(j);

   return 0;
}
```

Subactivity 20.a.iv

Adapt the `displayRow` function so that it can be used to display a row comprised of any character, not just an asterisk. Hint: You will need an additional formal parameter of type `char`.

Then test your function with the program of Subactivity 20.a.iii. Make it display a triangle of 'at' characters (`@`).

Subactivity solution

```
//Display a triangle of 'at' characters
#include <iostream>
using namespace std;

void displayRow(int n, char c)
{
   for (int i = 1; i <= n; i++)
      cout << c;
   cout << endl;
}

int main( )
{
   int m;

   cout << "Enter a positive integer: ";
   cin >> m;


   for (int j = 1; j <= m; j++)
      displayRow(j, '@');
```

```
    return 0;
}
```

Make sure that you did the following in your solution:

- gave a name to the formal parameter of type `char` (we called ours `c`)
- used the parameter (`c`) instead of `'*'` in the `cout` statement
- added the literal `'@'` as an actual parameter in the call of function `displayRow`.

Note the two actual parameters in the function call: `j` is an `int` variable, but `'@'` is a literal (`char`) value. We can also use a complicated expression for an actual parameter. The only restriction is that the types of the actual parameters must match the types of the formal parameters.

Subactivity 20.a.v

Now you are almost ready to tackle the main activity. Read the problem statement again. Then, in your study notebook, write down an algorithm for this problem. In other words, write down the steps that will need to be performed.

Subactivity solution

We identified the following steps.

1. Input the first value

2. While the value is not negative:

2.a Call function displayRow to display the appropriate number of asterisks

2.b Add the value to a total

2.c Input the next value

3. Calculate the average of all the values

4. Call displayRow to display the appropriate number of plus signs.

Your steps might be slightly different, but they should cover all the aspects that we have covered above.

For instance, we input the first value before the loop, and then input the next value at the end of the loop to ensure that the negative value (entered to terminate the loop) is not displayed or included in the calculation of the total. If you input all the values inside (at the beginning) of the loop, you'll need an `if` statement to prevent this problem.

See what other adjustments need to be made to your algorithm if you missed anything else. Then translate your algorithm into a program to give the solution to the main activity.

Activity solution

Using our algorithm, the program would look something like this:

```
//Display a histogram for a series of values
#include <iostream>
using namespace std;

void displayRow(int n, char c)
{
   for (int i = 1; i <= n; i++)
      cout << c;
   cout << endl;
}

int main( )
{
   int value, total, many, average;

   total = 0;
   many = 0;

   cout << "Enter the values (negative to end):" << endl;
   cin >> value;

   while (value > 0)
   {
      total += value;
      many++;
      displayRow(value, '*');
      cin >> value;
   }

   average = total/many;
   cout << "The average is " << average << endl;
   displayRow(average, '+');

   return 0;
}
```

Discussion

Note that the above program does not deal with a number of possible problems:

- The user starts off with zero.
- The user enters a value greater than 80. In this case, the row of asterisks will go over to the next line.
- We have used integer division to calculate the average of the numbers. Integer division throws away any fraction.

The program will display strange behaviour if any of these problems occur. By all rights we ought to build checks into the program to prevent the user from entering incorrect values. We could also adapt the program to round off the average instead of throwing away the fraction part.

Important points in this lesson

Programming concepts

We can design a void function to receive one or more values from a calling statement, perform some processing on them and then display some output on the screen. We do this by specifying the types and names of the parameters in a list of formal parameters in the function header. The general layout of the header of a void function is

```
void FunctionName(FormalParameterList)
```

or in more detail

```
void FunctionName(Type1 FormalParameter1,
                  Type2 FormalParameter2,
                  :
                  TypeN FormalParameterN)
```

To cause the function to be executed, the values to be sent to the function need to be specified as actual parameters via a function call which acts as a statement on its own:

```
FunctionName(ActualParameterList);
```

or

```
FunctionName(ActualParameter1, ActualParameter2, ...,
             ActualParameterN);
```

When the function is called, the value of *ActualParameter1* is copied to *FormalParameter1*, the value of *ActualParameter2* to *FormalParameter2*, etc. C++ is very precise about this. The transfer of values between the actual and formal parameters always takes the order into account. In other words, the first actual parameter always goes to the first formal parameter, and so on. As a result, there must always be the same number of actual parameters in the calling statement as the number of formal parameters in the function header, and their types must match, otherwise C++ will give an error message.

We said above that the body of a void function does not contain a `return` statement. This is not strictly true. A void function may contain an empty `return` statement, namely `return;`. This will cause immediate termination of the function. We, however, do not use it.

Programming principles

We use void functions when we have some code that is repeated in our program, or when a group of statements belong together to perform a subtask, or when input or output has to be done.

Exercises

Exercise 20.1

Write a program to draw a frame of asterisks. The user must enter two integers representing the width and height of the frame. For example,

```
* * * * * * *
*           *
*           *
*           *
* * * * * * *
```

should be displayed for a width of 7 and a height of 5. Note that the asterisks forming the top and bottom rows of the frame are separated by single space characters.

Exercise 20.2

Write a program to draw a tree:

```
   *
  * *
 * * *
* * * *
   *
   *
```

Your program must be able to draw a tree of any size, determined by the user. The tree shown above resulted from an input of 4. Note that the height and width of the branches of the tree are 4. The length of the trunk should always be 2, no matter what the size of the tree.

Hint: This problem is trickier than it appears at first because the space before the first asterisk on each line is dependant on the size of the tree. We recommend that you use a function to display one line at a time. This function should have two parameters: one to specify the number of space characters to display before the first asterisk, and one to specify the number of asterisks.

Exercise 20.3

Fix the solution to Activity 20.a so that it deals with the problems mentioned at the end of the solution. Firstly, it should prevent the user from entering incorrect values. In other words, by means of loops the user should be asked repeatedly to enter a value until it is correct. Secondly, it should store the average as a floating point value, and round it off to an integer before displaying the appropriate number of plus signs.

Lesson 21

Reference parameters, part 1

Purpose of this lesson

Sometimes we want a function to change the values of variables that are declared in the `main` function. We want to give the function access to the variables so that it can refer to them and change their values. In Lesson 19 we saw one way of doing this - by using global variables. However, we hope that you were convinced by the examples and arguments that we presented there that using global variables is a bad idea.

Fortunately there is another way, namely to use *reference parameters*. In fact, all the parameters we have used so far were *value parameters*. As we have seen up to now, value parameters are a way of making it clear what values a function will need to do its work. Similarly, reference parameters are a way of making it clear what variables a function will need to refer to, to do its work, i.e. what variables it will change the values of.

If we change the value of a value parameter within a function it has no effect on any of the variables in the `main` function. When we want to be able to change the values of variables in the `main` function by means of a function, we use reference parameters.

Activity 21.a

Write a program to compute the average of three marks for two students, and determine (and display) the best average.

Use functions where needed, especially to save repetition of code.

Test yourself

Without using functions, you should be able to write this program on your own. The only problem is that it will be somewhat tedious, especially all the repetition. Never fear! One important reason for defining and using functions is to save repetition.

Subactivity 21.a.i

Start by writing out an algorithm for the main activity in your study notebook. In other words, write down the basic steps that will need to be performed by the program.

Subactivity solution

We came up with the following steps:

1. *Input three marks for the first student*
2. *Calculate the average of these marks*
3. *Input three marks for the second student*
4. *Calculate the average of these marks*
5. *Determine and display the best average*

Another possibility is

1. *Input three marks for the first student and calculate the average*
2. *Input three marks for the second student and calculate the average*
3. *Determine and display the best average*

We prefer the first algorithm, however, because it separates input from processing and output. As a general rule (but not a hard-and-fast rule), we try to keep the part of the program responsible for input separate from the part responsible for processing, separate from the part responsible for output.

Discussion

What we'd like to do now is to define functions that can be called for each of these steps, so that we can have a nice simple `main` function:

```
int main( )
{
   float mark1, mark2, mark3;
   float averageA, averageB;

   inputMarks(mark1, mark2, mark3);
   averageA = average(mark1, mark2, mark3);

   inputMarks(mark1, mark2, mark3);
   averageB = average(mark1, mark2, mark3);

   displayBest(averageA, averageB);

   return 0;
}
```

The subactivities below take you through writing the necessary functions.

Subactivity 21.a.ii

Adapt the following function to input three marks rather than one, and write it out in your study notebook:

```
void inputMark (float & markP)
{
   cout << "Enter a mark: ";
   cin >> markP;
}
```

Note the ampersand symbol `&` between the type and name of the parameter. This shows that it is a reference parameter. `markP` therefore refers to a variable used as the actual parameter in the call of function `inputMark`, and any change to `markP` in this function will change the value of the variable in the `main` function.

Subactivity solution

```
void inputMarks (float & mark1P, float & mark2P, float & mark3P)
{
   cout << "Enter three marks: ";
   cin >> mark1P >> mark2P >> mark3P;
}
```

This is the precise function required by the program for the main activity.

We could have used three `cout` statements and three `cin` statements, but we decided to keep things as simple as possible.

Discussion

We keep to our rule of using different names for the actual and formal parameters of a function. So with a reference parameter, the name of the formal parameter acts as an 'alias' or 'nickname' for the variable used as the actual parameter. It is a temporary alias, because the name only exists for the duration of the function.

We indicate this in a variable diagram as follows:

[mark1]|mark1P	*[mark2]|mark2P*	*[mark3]|mark3P*	*[averageA]*	*[averageB]*
?	*?*	*?*	*?*	*?*

This shows the state of memory when function `inputMarks` commences.

Note that all the names of the local variables in the `main` function are hidden. The three reference parameters have alternative names (indicated with the vertical bar `|`). Note also that we do not create new boxes and copy the values of the parameters. The actual parameters are just renamed to the names of their corresponding formal parameters.

When the function terminates, all three parameters should have values in them. The temporary names are removed, and all the original names are visible again.

Subactivity 21.a.iii

Now write the `average` function for the next step of the algorithm. You should be able to do this without any difficulty since it is like the functions we wrote in Lessons 18 and 19. You simply need to add the three marks together, divide by 3 and return the result.

Subactivity solution

```
float average (float mark1P, float mark2P, float mark3P)
{
   return (mark1P + mark2P + mark3P) / 3;
}
```

This is a value-returning function, so it has a return type (`float` instead of `void`) and a `return` statement. Note also that all three parameters are value parameters, not reference parameters. This is because the function does not need to change or refer to any variables in the `main` function. It simply needs three values to do its work.

Discussion

Here is something interesting to think about. What is the difference between using a reference parameter and returning a value? Aren't they both ways of changing a variable in the `main` function? For example, the statement

```
averageA = average(mark1, mark2, mark3);
```

is used to change the value of variable `averageA` in the `main` function.

The answer is that it is true; these two ways of doing things are often inter-changeable. (In the next subactivity, you are required to change function `average` so that it uses a reference parameter instead of returning a value.)

One situation where we don't have a choice is where the function needs to change more than one variable. A value-returning function can only return a single value. If the function needs to return a number of values, we have no choice but to use reference parameters. For example, the function call

```
inputMarks(mark1, mark2, mark3);
```

is used to store three values, one in each of the variables `mark1`, `mark2` and `mark3`. We can't use a value-returning function to do this.

Subactivity 21.a.iv

Rewrite function `average` so that it is a void function. Add a reference parameter so that the value of the variable used as the fourth actual parameter of a function call can be changed by the function.

Subactivity solution

```
void calcAverage (float mark1P, float mark2P, float mark3P, float & averageP)
{
   averageP = (mark1P + mark2P + mark3P) / 3;
}
```

Note the following about this function:

- It is void so it does not have a `return` statement.
- It has four parameters. The first three are value parameters, and the last one is a reference parameter.
- It assigns a new value to the reference parameter (the same value that was returned by the value-returning function).
- We have changed the name of the function from `average` to `calcAverage`. This is because it doesn't return an average any more, but calculates and stores the average in a variable. This function will be called as a statement on its own, e.g.

```
calcAverage(mark1, mark2, mark3, averageA);
```

Discussion

This function illustrates some important differences between void and value-returning functions.

- Void functions do not have a return type and do not need a `return` statement.
- Void functions can have one or more reference parameters (although all their parameters can also be value parameters). C++ does allow value-returning functions to have reference parameters, but we follow the convention of never using reference parameters with value-returning functions. The combination can cause horrible confusion.
- We generally change the values of reference parameters (if any) in a void function. If we don't change their values, they should rather be value parameters.
- We choose the names of void and value-returning functions carefully. The name of a void function should describe what it does, so the name is generally a verb or contains a verb (e.g. `displayRow` or `calcAverage`). The name of a value-returning function should describe the value that it returns, so the name is generally a noun (e.g. `average`).

Subactivity 21.a.v

Now write the last function, namely `displayBest`.

Subactivity solution

For this we use a void function with two value parameters. Here are two solutions:

```
void displayBest(float averageAP, float averageBP)
{
   if (averageAP > averageBP)
      cout << "The best average is " << averageAP << endl;
   else
      cout << "The best average is" << averageBP << endl;
}
```

or

```
void displayBest(float averageAP, float averageBP)
{
   float best = averageAP;
   if (averageBP > averageAP)
      best = averageBP;
   cout << "The best average is " << best << endl;
}
```

Even though this second version introduces an additional local variable, we prefer it for two reasons: It separates processing from output, and it avoids repetition of the same output message.

Discussion

One question that you might be asking is whether it is necessary to use a function at all for this code. Wouldn't it be easier just to put the code in the `main` function? There really isn't a hard-and-fast rule to answer this question. It is more a question of style and personal preference. We like the idea of putting this code in a function, because it makes the `main` function nice and neat, and easy to read. All the gory detail is hidden away in the other functions.

Finally, you might be wondering what would happen if we made the parameters of this function reference parameters (by inserting `&` between their types and their names). Try it and see. You should notice that it doesn't make any difference at all. However, it is not good programming practice to only use reference parameters. Firstly, this could allow inadvertent errors to creep in that are difficult to find (e.g. if the function changes the value of a parameter when it shouldn't). Secondly, the parameters should tell other people who read your code which of the parameters will be changed by the function and which won't. By only using reference parameters, this information is lost and the reader will have to study the body of the function carefully to see whether it does in fact change the values of parameters or not.

Subactivity 21.a.vi

Now put all the pieces together to form a program. Use the solutions of Subactivities 21.a.ii, 21.a.iii and 21.a.v together with the `main` function given in the **Discussion** section at the end of Subactivity 21.a.i to write the entire program. Doing this will give you the solution to Activity 21.a.

Activity solution

```
//Input three marks for two students
```

```
//and calculate and display the best average
#include <iostream>
using namespace std;

void inputMarks(float & mark1P, float & mark2P, float & mark3P)
{
   cout << "Enter three marks: ";
   cin >> mark1P >> mark2P >> mark3P;
}

float average(float mark1P, float mark2P, float mark3P)
{
   return (mark1P + mark2P + mark3P) / 3;
}

void displayBest(float averageAP, float averageBP)
{
   float best = averageAP;
   if (averageBP > averageAP)
      best = averageBP;
   cout << "The best average is " << best << endl;
}

int main( )
{
   float mark1, mark2, mark3;

   float averageA, averageB;

   inputMarks(mark1, mark2, mark3);
   averageA = average(mark1, mark2, mark3);

   inputMarks(mark1, mark2, mark3);
   averageB = average(mark1, mark2, mark3);

   displayBest(averageA, averageB);

   return 0;
}
```

Activity 21.b

For this activity, change the solution to Activity 21.a (shown above) to compute the average of four marks for three students.

Test yourself

First think how many changes you would have to make if there were no functions - if everything was in the `main` function. There would be tremendous duplication and the program would be very long. This activity is intended to convince you of the usefulness of functions to save repetition of code.

Now try to write this program on your own. Ask yourself what changes need to be made. How should the function headers and the function calls be changed to include an additional mark? How should the `main` function be changed to compute the average mark for a third student? Try this now by editing the program that you typed in for the previous activity.

If you were able to change the program, compare your answer to the one given in the activity solution below. If your answer is very different, or if you do not get the new program to work, work through the subactivities below.

Subactivity 21.b.i

Our first step is to change function `inputMarks` to provide for four marks. Try this on your own before you read further.

Subactivity solution

```
void inputMarks(float & mark1P, float & mark2P, float & mark3P, float & mark4P)
{
   cout << "Enter four marks: ";
   cin >> mark1P >> mark2P >> mark3P >> mark4P;
}
```

As easy as jam.

Subactivity 21.b.ii

Now change function `average` to provide for four marks.

Subactivity solution

```
float average(float mark1P, float mark2P, float mark3P, float mark4P)
{
   return (mark1P + mark2P + mark3P + mark4P) / 4;
}
```

Also very straightforward.

Subactivity 21.b.iii

Now change the `displayBest` function to determine and display the best average.

Hint: Introduce a local variable to help you.

Subactivity solution

```
void displayBest(float averageAP, float averageBP, float averageCP)
{
   float best = averageAP;
   if (averageBP > best)
      best = averageBP;
   if (averageCP > best)
      best = averageCP;
   cout << "The best average is " << best << endl;
}
```

Subactivity 21.b.iv

You can now change the `main` function to call the three changed functions. Remember to add the necessary statements to compute the average for a third student.

Subactivity solution

Did you try this on your own? What statements did you change? First of all you need to declare additional variables for the fourth mark and the average for the third student, namely `mark4` and `averageC`.

Then you need to include variable `mark4` in the calls to `inputMarks`:

```
inputMarks(mark1, mark2, mark3, mark4);
```

Next, you need to include `mark4` in the calls to `average`:

```
averageA = average(mark1, mark2, mark3, mark4);
```

Then you need to add the following statements to deal with a third student:

```
inputMarks(mark1, mark2, mark3, mark4);
averageC = average(mark1, mark2, mark3, mark4);
```

Finally, you need to change the call of `displayBest` to take all three averages as parameters:

```
displayBest(averageA, averageB, averageC);
```

You should now have made all the changes to the program needed to meet the requirements of the main activity.

Activity solution

```
//Input four marks for three students
//and calculate and display the best average
```

```
#include <iostream>
using namespace std;

void inputMarks(float & mark1P, float & mark2P, float & mark3P, float & mark4P)
{
   cout << "Enter four marks: ";
   cin >> mark1P >> mark2P >> mark3P >> mark4P;
}

float average(float mark1P, float mark2P, float mark3P, float mark4P)
{
   return (mark1P + mark2P + mark3P + mark4P) / 4;
}

void displayBest(float averageAP, float averageBP, float averageCP)
{
   float best = averageAP;
   if (averageBP > best)
      best = averageBP;
   if (averageCP > best)
      best = averageCP;
   cout << "The best average is " << best << endl;
}

int main( )
{
   float mark1, mark2, mark3, mark4;
   float averageA, averageB, averageC;

   inputMarks(mark1, mark2, mark3, mark4);
   averageA = average(mark1, mark2, mark3, mark4);

   inputMarks(mark1, mark2, mark3, mark4);
   averageB = average(mark1, mark2, mark3, mark4);

   inputMarks(mark1, mark2, mark3, mark4);
   averageC = average(mark1, mark2, mark3, mark4);

   displayBest(averageA, averageB, averageC);

   return 0;
}
```

We have only needed to change a few lines of code and add a few more to get this program to deal with more marks and more students. This was made possible because of the use of functions.

Discussion

We have used a lot of different terminology to refer to the different kinds of parameters. This may be a good time to explain them all again.

We say that a function call (which is generally in the `main` function but can also be in another function) contains an *actual parameter list*. This list includes all parameters which are sent to the function being called. All variables in this list must be declared in the calling function.

The called function contains a *formal parameter list* in the function header. None of the parameters in this list need to be declared separately in the function since the formal parameter list is a kind of declaration statement itself.

The number of parameters in the actual parameter list must agree with the number of parameters in the formal parameter list although the parameter names need not be the same. (We generally use different names, even though the compiler will allow the same names.) The parameter types must also match. For example, a `char` actual parameter cannot be passed to a `float` formal parameter.

Formal parameters which are used to get copies of values from the calling function are called *value parameters*. Formal parameters which are used to change variables in the calling function are called *reference parameters*. A reference parameter temporarily renames an existing variable instead of creating a new variable with a copy of the value.

But what is the difference between a variable and a parameter? A variable is a programmer-defined name used to store a value in the program. A parameter can be a variable but is referred to as a parameter when it is included in a list enclosed within parentheses, like the list in a function call or a function header.

Variables which are declared within a particular function are called *local variables* and are only active while the function they are declared in is executing. The parameters defined in the formal parameter list are also local to the particular function. In other words, outside the particular function, local variables and formal parameters of that function are inaccessible.

Activity 21.c

Write a program to input a large number of seconds, and to work out how many weeks, days, hours, minutes and seconds the number represents. For example, the output for the value 654321 should be:

```
654321 seconds represents
1 weeks, 0 days, 13 hours, 45 minutes and 21 seconds
```

Test yourself

As you no doubt suspect, there will be a fair amount of repetition in this program. In general, we use a loop to deal with repetition. But to use a loop you must be doing exactly the same thing over and over. Here we have a problem, because we need to do a slightly different thing each time. In particular, since there are 60 seconds in a minute, 60 minutes in an hour, 24 hours in a day and 7 days in a week, we need to divide the remaining value by a different value each time.

There is another way to deal with repetition, namely to write a function that contains the common code, and that deals with differences by means of its parameters. Then you just call the function as many times as needed, each time with the necessary parameters.

With this hint, see if you can write the program using such a function.

Subactivity 21.c.i

In your study notebook, write a function that takes an integer representing a number of seconds as parameter, and returns two integers, one representing the number of minutes and one representing the number of remaining seconds.

Subactivity solution

```
void calcMinutes(int secondsP, int & minutesP, int & remSecondsP)
{
   minutesP = secondsP / 60;
   remSecondsP = secondsP % 60;
}
```

Discussion

We can't get a function to return two (or more) values as the value of the function. A function can either be void or return a single value. It would be possible to make the above function return one of the two values as the value of the function, and the other as a reference parameter, but this is considered poor programming style. In fact, sometimes you can get yourself into a nasty mess by using reference parameters together with a value-returning function.

The general rule is if you need a function to return more than one value, make it a void function and use two (or more) reference parameters.

Subactivity 21.c.ii

Now write a program that inputs an integer representing a large number of seconds, uses the function above to calculate the number of minutes and seconds it represents, and displays the result.

Subactivity solution

```
//Calculate minutes and seconds
#include <iostream>
using namespace std;

void calcMinutes(int secondsP, int & minutesP, int & remSecondsP)
{
   minutesP = secondsP / 60;
   remSecondsP = secondsP % 60;
}

int main( )
{
   int seconds;
   int minutes;
   int remSeconds;

   cout << "Enter a large (positive) integer: ";
```

```
   cin >> seconds;

   calcMinutes(seconds, minutes, remSeconds);

   cout << seconds << " seconds represents" << endl;
   cout << minutes << " minutes and " << remSeconds << " seconds" << endl;

   return 0;
}
```

Subactivity 21.c.iii

Rewrite the above function to make it more general. In particular, adapt it so that it doesn't assume we are working with minutes and seconds, but rather with any two units that have some ratio. Include the ratio as an additional parameter so that the function can be called with the following statement (for example):

```
calcRemainder(days, 7, weeks, remDays);
```

where `days`, `weeks` and `remDays` are all integer variables.

Subactivity solution

```
void calcRemainder(int unit1, int ratio, int & unit2, int & remUnit1)
{
   unit2 = unit1 / ratio;
   remUnit1 = unit1 % ratio;
}
```

Note how we have changed the names of all the parameters to general names that can be used to represent any units.

Subactivity 21.c.iv

Now change the program you wrote to use this function (still for minutes and seconds only).

Subactivity solution

```
//Calculate minutes and seconds
#include <iostream>
using namespace std;

void calcRemainder(int unit1, int ratio, int & unit2, int & remUnit1)
{
   unit2 = unit1 / ratio;
   remUnit1 = unit1 % ratio;
}

int main( )
{
   int seconds, minutes;
```

```
   int remSeconds;

   cout << "Enter a large (positive) integer: ";
   cin >> seconds;

   calcRemainder(seconds, 60, minutes, remSeconds);

   cout << seconds << " seconds represents" << endl;
   cout << minutes << " minutes and " << remSeconds << " seconds" << endl;

   return 0;
}
```

With this framework, you should be able to expand the program to deal with all the units. If you feel unsure about dealing with all the units at once, add one at a time and test your program.

Activity solution

```
//Calculate weeks, days, hours, minutes and seconds
#include <iostream>
using namespace std;

void calcRemainder(int unit1, int ratio, int & unit2, int & remUnit1)
{
   unit2 = unit1 / ratio;
   remUnit1 = unit1 % ratio;
}

int main( )
{
   int seconds, minutes, hours, days, weeks;
   int remSeconds, remMinutes, remHours, remDays;

   cout << "Enter a large (positive) integer: ";
   cin >> seconds;

   calcRemainder(seconds, 60, minutes, remSeconds);
   calcRemainder(minutes, 60, hours, remMinutes);
   calcRemainder(hours, 24, days, remHours);
   calcRemainder(days, 7, weeks, remDays);

   cout << seconds << " seconds represents" << endl;
   cout << weeks << " weeks, " << remDays << " days, ";
   cout << remHours << " hours, " << remMinutes << " minutes";
   cout << " and " << remSeconds << " seconds" << endl;

   return 0;
}
```

Important points in this lesson

Programming concepts

This lesson introduced reference parameters. They are used to allow a called function to change the values of variables declared in the `main` function. When a function changes the value of one of its reference parameters, it actually changes the value of a variable declared in the `main` function that is used as the corresponding actual parameter. (As you will remember from the previous lesson, value parameters contain copies of values sent from the `main` function to the called function. So any changes made to value parameters within a function have no effect on any variables in the `main` function.)

To specify a reference parameter, we use the `&` symbol between the type and the name of the parameter in the formal parameter list in the function header.

```
void FunctionName(ParameterType & ParameterName)
```

A function can have more than one reference parameter, or a mixture of value and reference parameters.

Programming principles

The name of a void function should describe what it does, so the name is generally a verb or contains a verb (e.g. `displayRow` or `calcAverage`). The name of a value-returning function should describe the value that it returns, so the name is generally a noun (e.g. `average`) or an adjective (e.g. `smallest`).

Sometimes it is difficult to decide whether a function should be a value-returning function or a void function with a reference parameter. A general rule of thumb is to ask yourself what is the most important: what the function does or what it returns? If what the function returns is more important than what it does, it should probably be a value-returning function. On the other hand, if the function contains lot of processing or does any input or output, a void function is preferable.

Also, if a function needs to return more than one value, make it a void function and use two or more reference parameters. Don't use reference parameters with a value returning function.

Don't use a reference parameter if a value parameter would work. In other words, if the function does not change the value of a parameter, don't make it a reference parameter.

Exercises

Exercise 21.1

Consider the following program:

```
// Simulates throwing dice until a total of 7 is thrown
#include <iostream>
using namespace std;

int totalDice( )
{
   int die1 = rand( )%6 + 1;
   int die2 = rand( )%6 + 1;
   cout << "Throw: " << die1 << " and " << die2 << endl;
   return die1 + die2;
```

```
}

int main( )
{
   srand(time(0));

   int count = 0;
   while (totalDice( ) != 7)
      count++;

   cout << "It took " << count << " throws ";
   cout << "before 7 was thrown." << endl;

   return 0;
}
```

The `totalDice` function above is a value-returning function but also does output (it displays the values of the separate dice). As explained in the **Important points** section above, we prefer not to do input and output in value-returning functions.

Change `totalDice` to a void function with a single reference parameter. Instead of returning the total, it must store the total in its reference parameter.

Remember that you should also change the name of your function, as explained in the **Important points** section.

Also change the `main` function so that it calls your new function correctly.

Exercise 21.2

Redo Exercise 20.3, using void functions to input and validate the relevant values.

Exercise 21.3

Write a function that receives the radius of a circle as a value parameter and calculates the area and circumference of the circle, storing the results in reference parameters. Write a `main` function to call this function. Draw up at least three test cases and test your function.

The formulas involved are:

$$area = \pi.radius^2$$

$$circumference = 2.\pi.radius$$

Exercise 21.4

Rewrite the function that you wrote for Exercise 21.3 above as two value returning functions: one to return the area of a circle (given its radius) and one to return the circumference.

Lesson 22

Reference parameters, part 2

Purpose of this lesson

We have seen how to use value parameters to send values from the `main` function to a called function. We have also seen how to use reference parameters to allow a called function to change the values of variables declared in the `main` function.

However, the functions that used the reference parameters never assumed that there were any values in those variables. In fact, we used the functions to simply initialise variables for the `main` function.

Sometimes, though, we want a function to update the values of variables that it gets from the `main` function. In other words, the function uses the values already stored in its reference parameters to calculate a new value for them. These changes are reflected in the variables used as actual parameters in the function call.

Activity 22.a

Patricia sells a variety of crafty items at the local flea market. In order to keep track of her income and expenditure, she wants a program that can add up the total value of her stock.

Write a program that inputs the stock of the items that she sells, and calculates and outputs the total stock value. The name and price of each item must be displayed for her to know what to input.

Here is an example of the user interaction:

```
Enter the stock of each item
Plain candles: R8.00
Stock: 22
Scented candles: R10.00
Stock: 14
:
Hand-painted Ms: R23.50
Stock: 9
Total value of the stock is R534.20
```

Note, the names of the items and their prices have to be built into the program. They are not input from the keyboard.

Test yourself

As in Activity 21.c, we have the problem of requiring repetition where we can't use a loop. In this case, a different message (in the form of the name and price) has to be displayed for each item.

As we saw in Activity 21.c, one use of functions is to reduce repetition of code. See if you can think how to use a function to save having to repeat code for each item. The trick is to put as much common code in the function as possible, and then call it over and over with the necessary parameters.

Before you tackle this problem, consider the following interesting use of a reference parameter:

Subactivity 22.a.i

Consider the following function that uses a global constant `VAT_RATE`:

```
void addVAT(float & priceP)
{
   priceP += priceP * VAT_RATE;
}
```

How does the use of the reference parameter `priceP` differ from how we used reference parameters in the previous lesson?

Write a short program that uses this function. It should input a price, then call this function and finally output the updated price.

Subactivity solution

The difference is that function `addVAT` assumes that parameter `priceP` already has a value. If `priceP` does not have a meaningful value when the function commences, its final value will also be garbage.

Here is a program that uses `addVAT`:

```
//Input a price, add VAT and display the updated price
#include <iostream>
using namespace std;

const float VAT_RATE = 0.14;

//Update the price by adding VAT
void addVAT(float & priceP)
{
   priceP += priceP * VAT_RATE;
}

int main( )
{
   float price;
```

```
    cout << "Enter the price of the item: ";
    cin >> price;

    addVAT(price);

    cout << "The price with VAT included is " << price << endl;

    return 0;
}
```

Discussion

Reference parameters can either be used to initialise variables passed to a function as parameters (as we saw in the previous lesson), or they can be used to update the values of variables passed as parameters (as shown above). If they are used to update the values of variables, the calling function has the responsibility of initialising such variables.

Now to get back to the main activity:

Subactivity 22.a.ii

As stated in the **Test yourself** section above, we want to use a function to deal with the repetition in the program. Decide what the function will do. In particular, write a description (to be placed as a comment before the function header) and then write the function header including all the parameters. Do this in your study notebook.

Subactivity solution

There are a few possibilities. Here is the description of one such function:

```
//Input the stock of an item and calculate the stock value using the price
```

For this function, we need a value parameter to send the price of the item, and a reference parameter to send back the stock value of the item. Think about it: the stock number can be a local variable of the function. (If you need to adjust your function with this information, do so now!)

The header for this function will therefore look like this:

```
void inputAndCalc(float priceP, float & stockValP)
```

Note that we use a void function rather than a function that returns the value of the stock as its return value since the function does input. Our general rule is that if a function does input, it should be a void function.

One of the problems with the function is that the code to display the different message for each item (i.e. its name and price) is outside the function. By using a string parameter, we can send the name of the item to the function, and it can display the prompt message. The description of this function will be as follows:

```
//Display the name and price of an item, input the stock, and calculate the
//stock value using the price
```

The header of this function will be:

```
void inputAndCalc(string nameP, float priceP, float & stockValP)
```

Note how we have two value parameters and a reference parameter in this function.

Another problem with the function in its current form is that it doesn't add up the total stock value. It leaves the job to the `main` function where a calculation in an assignment statement will have to be performed between each call of the function.

To solve this problem, we need to send the stock total as a parameter to be updated by the function (as the `addVAT` function did in the previous subactivity).

The description of this function would be

```
//Display the name and price of an item, input the stock,
//calculate the stock value using the price and add it to the stock total
```

If you think about it, we no longer need to send the stock value as a parameter to this function. The `main` function doesn't need to work with this variable or have any access to it, so it can be declared as a local variable of the function. The header for our final version of the function will therefore be

```
void inputAndCalc(string nameP, float priceP, float & stockTotalP)
```

Subactivity 22.a.iii

Write the body of the `inputAndCalc` function as specified above, also in your study notebook.

Subactivity solution

```
//Display the name and price of an item, input the stock,
//calculate the stock value using the price and add it to the total
void inputAndCalc(string nameP, float priceP, float & stockTotalP)
{
   int stock;
   float stockVal;
   cout << nameP << ": R" << priceP << endl;
   cout << "Stock: ";
   cin >> stock;
   stockVal = priceP * stock;
   stockTotalP += stockVal;
}
```

You should already be convinced of the usefulness of such a function. Imagine having to type in all this code for each item!

Subactivity 22.a.iv

Write the `main` function that uses this function to form the answer to the main activity. Use about seven different items - you will need to think up the names of the items that Patricia sells!

Activity solution

```
//Calculates total value of stock
#include <iostream>
#include <string>
using namespace std;

//Display the name and price of an item, input the stock,
//calculate the stock value and add it to the total
void inputAndCalc(string nameP, float priceP, float & stockTotalP)
{
   int stock;
   float stockVal;
   cout << nameP << ": R" << priceP << endl;
   cout << "Stock: ";
   cin >> stock;
   stockVal = priceP * stock;
   stockTotalP += stockVal;
}

int main( )
{
   float stockTotal = 0;

   cout << "Enter the stock of each item" << endl;
   inputAndCalc("Plain candles", 8.00, stockTotal);
   inputAndCalc("Scented candles", 10.00, stockTotal);
   inputAndCalc("Dream-catchers", 19.00, stockTotal);
   inputAndCalc("Bead place mats", 11.50, stockTotal);
   inputAndCalc("Bead coasters", 5.00, stockTotal);
   inputAndCalc("Incense holders", 6.40, stockTotal);
   inputAndCalc("Hand-painted Ms", 23.50, stockTotal);

   cout << "Total value of the stock is R" << stockTotal << endl;

   return 0;
}
```

Note how we initialise `stockTotal` at the beginning of the program, and how it is updated each time the function is called.

Important points in this lesson

Programming concepts

In the previous lesson we saw that functions can be used to initialise variables declared in the main function by means of reference parameters. In this lesson we saw that a reference parameter can also be used to update a variable, i.e. use its old value to calculate a new value.

Whenever the value of a reference parameter is changed within a function, the value of the corresponding variable in the main function also changes. If the value of a value parameter is changed, no values of variables

in the main function change.

Programming principles

If a function should update the value of a variable, use a void function with a reference parameter. If a function should use the value of a variable and calculate a separate value, leaving the variable unchanged, use a value returning function with a value parameter.

Think carefully about what should be local variables, what should be value parameters and what should be reference parameters.

Exercises

Exercise 22.1

(i) Write a function named `increment15`, with a single integer reference parameter, to add 15 to the variable that it takes as parameter. Document the function by adding a comment statement.

(ii) Write a simple `main` function and use appropriate test cases to test this function.

Exercise 22.2

Consider the following function:

```
void swap(int & n1, int & n2)
{
   int temp;
   temp = n1;
   n1 = n2;
   n2 = temp;
}
```

(i) Add a comment to the function to tell a reader what the function does.

(ii) Write a program that inputs values into two variables, calls function `swap` with the two variables as parameters, and then displays the values of the variables after the function has finished, showing the new order. Test the function appropriately.

Exercise 22.3

Write a function called `rotate` (similar to function `swap` in Exercise 22.2(i)) that takes three integer variables and rotates their values.

Adapt the program you wrote for Exercise 22.2(ii) to test this function.

Lesson 23

Variable diagrams (again!)

Purpose of this lesson

We introduced variable diagrams in Lesson 5, and have used them in various lessons since then to see how the values of variables change, and if the program does what is expected. In this lesson, we concentrate on variable diagrams again, this time using them with functions and parameters.

In particular, we use variable diagrams to clarify the differences between value and reference parameters, and to illustrate which part of a program can refer to which variables. Since this lesson brings together the work already covered in the previous few lessons, it is important that you work through this lesson thoroughly, making sure that you understand each set of variable diagrams.

Activity 23.a

Study the program below and, by drawing variable diagrams, show the output, given an input value of 4. In your answer, remember to show the value of each variable and each parameter for each function.

```
1  //Squared numbers
2  #include <iostream>
3  using namespace std;
4
5  void calcSquare1(int x, int y)
6  {
7     y = x * x;
8  }
9
10 void calcSquare2(int x, int & y)
11 {
12    y = x * x;
13 }
14
15 void calcSquare3(int & x, int & y)
16 {
17    y = x * x;
18 }
19
20 int main( )
21 {
22    int x, y;
23    cout << "Enter a number: ";
24    cin >> x;
```

```
25      calcSquare1(x, y);
26      calcSquare2(x, y);
27      calcSquare3(y, x);
28      calcSquare2(y, x);
29      calcSquare1(x, y);
30      cout << "x = " << x << ", y = " << y << endl;
31      return 0;
32  }
```

Beware! We have purposely used the same names for variables in the `main` function as for parameters in the other three functions, and we have swapped them around for some calls to try and confuse you!

Test yourself

Compare your variable diagrams to the activity solution given later in this lesson. If your diagrams differ from ours, then work through the subactivities and the discussions below.

Subactivity 23.a.i

Start by identifying (in your study notebook) what kinds of parameters are used for each function. In other words, determine which parameters are value parameters and which are reference parameters. Doing this identifies when values of variables in the `main` function change because of changes made in the functions.

Subactivity solution

From the previous lessons you will remember that if an ampersand (`&`) is used between a parameter type and its name, it is a reference parameter, if not, it is a value parameter.

Function	*Value parameters*	*Reference parameters*
calcSquare1	*x, y*	*-*
calcSquare2	*x*	*y*
calcSquare3	*-*	*x, y*

Discussion

Before starting the variable diagrams, let's call to mind the conventions we use for variable diagrams:

- A question mark *?* shows an uninitialised value for a variable.
- The notation *25* $\rightarrow$ *5* means that execution jumps from line 25 to line 5.
- We use square brackets *[ ]* around the name of a variable to show that it is inaccessible while the current function is being executed.

Subactivity 23.a.ii

Now start drawing variable diagrams for the `main` function, from where the program starts executing up to where the first call to `calcSquare1` terminates.

Subactivity solution

Line 22: `int x, y;`

The program itself always starts executing at the beginning of the `main` function, which in this case is the declaration in line 22.

The variable declaration in this line brings two variables, `x` and `y` into existence. We do not yet know what their values are, and so we show them with question marks:

Line 22:

x	y
?	?

Line 23: `cout << "Enter a number:  ";`

This displays a prompt message asking the user to enter a number. None of the variables change, so we don't make any changes to the variable diagrams.

Line 24: `cin >> x;`

This statement inputs the value for variable `x` from the keyboard. According to the question, the input value must be 4.

Line 24:

x	y
4	?

Line 25 → 5: `calcSquare1(x, y);`

From the definition of `calcSquare1` (line 5), we see that this function expects to receive two integers as value parameters, also called `x` and `y`. We therefore create two more boxes, label them `x` and `y` and copy the values of the actual parameters to the corresponding formal parameters. Since we do not know the value of the second actual parameter (`y`) in the `main` function, we also do not know the value of the second formal parameter (`y`) in function `calcSquare1`.

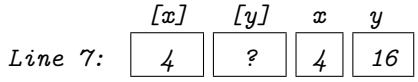
Line 25 → 5:

[x]	[y]	x	y
4	?	4	?

Note that the `x` and `y` variables declared in the `main` program are inaccessible now, so we put square brackets around their names.

Line 7: `y = x * x;`

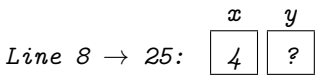
This tells the computer to multiply the value of `x` by itself, and to place that value in `y`. Now we know the value of `y` in `calcSquare1`, so we change its value in the variable diagrams.



The square brackets around the names of `x` and `y` in the `main` function mean that their values are inaccessible while `calcSquare1` is being executed. Since we cannot access the variables in the `main` function, the value of `y` in the `main` function is still unknown.

Line 8 → 25: }

The close brace in line 8 indicates that function `calcSquare1` has come to an end. Its value parameters are destroyed.



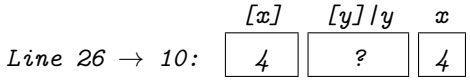
Since `calcSquare1` uses only value parameters, no variables have been updated in the `main` function. So, we lose the value of `y` in `calcSquare1`, and the value of `y` in the `main` function is still unknown by the end of line 25.

Subactivity 23.a.iii

To see how your skill in drawing variable diagrams is coming along, draw a series of variable diagrams for the next function call (line 26) and the execution of function `calcSquare2`.

Subactivity solution

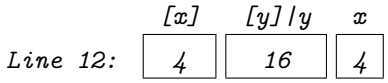
Line 26 → 10: `calcSquare2(x, y);`



Note that we only create an extra box for parameter `x` because it is a value parameter. The value of the actual parameter is copied to it. We do not create an extra box for parameter `y` because it is a reference parameter. We simply give an alternative name to the variable passed as the actual parameter (in this case also `y`). Both the names `x` and `y` of the `main` function are hidden for the duration of `calcSquare2`.

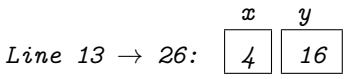
Line 12: `y = x * x;`

As in line 7, `x` is multiplied by itself, and the answer is stored in `y`. However, for `calcSquare2`, `y` is a reference parameter referring to `y` in the `main` function. Thus, `y` in the `main` function is also updated to 16.



Line 13 → 26: }

The function terminates and all value parameters are destroyed. The names of all local variables of the `main` function are accessible again:



Subactivity 23.a.iv

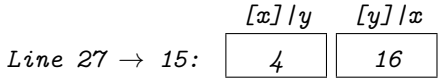
Now you are ready to do the next function call (line 27) in your study notebook.

Subactivity solution

Line 27 → 15: `calcSquare3(y, x);`

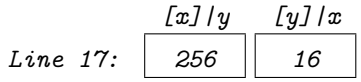
This is a call to function `calcSquare3`. But there is an important difference between line 27, and lines 26 and 25. We have changed the order of the actual parameters in the function call.

Another important thing to notice is that both formal parameters of `calcSquare3` are reference parameters. We therefore do not create any new boxes or copy any values, because neither of the parameters are value parameters. The names of variables `x` and `y` in the `main` function are hidden, but their memory positions get new (temporary) names.



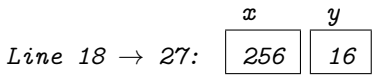
Line 17: `y = x * x;`

As before, `x` is multiplied by itself, and the answer is placed in `y`:



Line 18 → 27: }

The function terminates and we return to where it was called in the `main` function. The temporary names are removed and the original names are visible again:



Subactivity 23.a.v

Finally, draw variable diagrams for the next two function calls in your study notebook.

Subactivity solution

Line 28 → 10: `calcSquare2(x, y);`

	[x]/y	[y]	x
Line 28 → 10:	256	16	16

Once again, we only create an extra box for parameter `x` because it is a value parameter. The value of the actual parameter (`y`) is copied to it. We do not create an extra box for parameter `y` because it is a reference parameter. We simply give an alternative name to the variable passed as the actual parameter (in this case `x`). Both the names `x` and `y` of the `main` function are hidden for the duration of `calcSquare2`.

Line 12: `y = x * x;`

	[x]/y	[y]	x
Line 12:	256	16	16

The value of parameter `y` was 256, and this is replaced with 256!

Line 13 → 28: }

	x	y
Line 13 → 28:	256	16

Line 29 → 5: `calcSquare1(x, y);`

As before, we create two more boxes, label them `x` and `y` and copy the values of the actual parameters to the corresponding formal parameters.

	[x]	[y]	x	y
Line 29 → 5:	256	16	256	16

Line 7: `y = x * x;`

	[x]	[y]	x	y
Line 7:	256	16	256	65536

Line 8 → 29: }

	x	y
Line 8 → 29:	256	16

Since `calcSquare1` uses only value parameters, no variables have been updated in the `main` function. So, we lose the value of `y` calculated in `calcSquare1`.

Activity solution

We do not repeat all the sets of variable diagrams again here. They are given and explained in the subactivities above.

Working from the final set of variable diagrams, we can see that with a program input of 4, the output in line 30 will be:

```
x = 256, y = 16
```

Activity 23.b

Using variable diagrams if necessary, work out how the output of the program given in Activity 23.a will change if

- we swap the statements in lines 25 and 26?
- we swap the statements in lines 26 and 27?

Subactivity 23.b.i

Let us start by writing out the new main function if we swap the statements in lines 25 and 26:

```
20 int main( )
21 {
22    int x, y;
23    cout << "Enter a number: ";
24    cin >> x;
25    calcSquare2(x, y);
26    calcSquare1(x, y);
27    calcSquare3(y, x);
28    calcSquare2(y, x);
29    calcSquare1(x, y);
30    cout << "x = " << x << ", y = " << y << endl;
31    return 0;
32 }
```

Now, see if you can work out what the values of variables `x` and `y` (in the `main` function) after the call to `calcSquare1` in line 26. You can use variable diagrams if you like, but the answer is actually obvious. Can you see why?

Subactivity solution

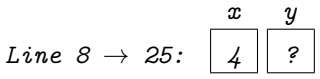
Instead of drawing variable diagrams to determine the answer, you can work out theoretically what the effect of calling `calcSquare2` first and then `calcSquare1` will be. Note that `calcSquare1` has two value parameters, and so, because it destroys its two parameters when it terminates, it does not affect `x` or `y` in the `main` function. Since `calcSquare1` has no effect on the values of variables in the `main` function, it makes no difference whether we call `calcSquare1` before or after `calcSquare2`.

Subactivity 23.b.ii

For the second subactivity, we swap the statements in lines 26 and 27 of the original program. The revised `main` function becomes:

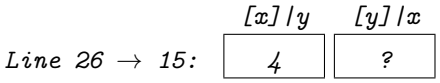
```
20 int main( )
21 {
22    int x, y;
23    cout << "Enter a number: ";
24    cin >> x;
25    calcSquare1(x, y);
26    calcSquare3(y, x);
27    calcSquare2(x, y);
28    calcSquare2(y, x);
29    calcSquare1(x, y);
30    cout << "x = " << x << ", y = " << y << endl;
31    return 0;
32 }
```

Draw a series of variable diagrams in your study notebook to see whether the outputs change. Start with the values after execution of line 25 as shown in the diagram below:

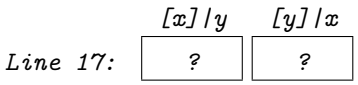


Subactivity solution

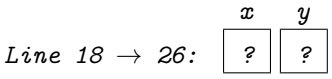
Line 26 → 15: `calcSquare3(y, x);`



Line 17: `y = x * x;`

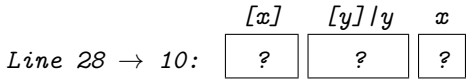


Line 18 → 26: }

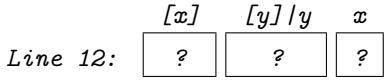


So, if we swap lines 26 and 27, the values of both variables become unititialised! The function calls that follow, can't fix the problem:

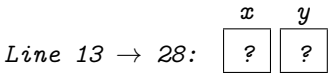
Line 28 → 10: `calcSquare2(x, y);`



Line 12: `y = x * x;`



Line 13 → 28: }



There are two morals to this story:

- The order of statements is important. In a few situations, statements that do not affect one another can be swapped around. In most cases however, certain things need to be done before others for a program to work correctly. Variable diagrams are a good way to check whether you've got the correct order of instructions.
- Leaving variables uninitialised can cause strange problems that are very difficult to solve. Always make sure that you initialise all variables properly before you use them. Usually it is straightforward to check whether a program tries to use a variable's value before a value has been stored in it (either by means of an input statement or an assignment statement). In such a case, initialise the variable either in its declaration, in a separate assignment statement, or by inputting its value (i.e. requesting a value from the user).

Important points in this lesson

Programming concepts

This lesson shows us how useful variable diagrams can be in investigating how a program works, and how they help to clarify several things, particularly about using functions with parameters:

- Parameters are transferred on the basis of their order, not by name. For example, the function call `calcSquare1(y, x)` will give a different result to `calcSquare1(x, y)`, even if the formal parameters of `calcSquare1` are called `x` and `y`.
- Value parameters make a copy of the values sent to them. Additional memory is required for this. They act as local variables for the duration of the function.
- Value parameters are destroyed when the function terminates, i.e. their values are lost. If we want a function to change the values of variables in the calling function, we must use reference parameters.
- The names of local variables in the calling function are hidden while another function is being called. Although their memory positions are still allocated, they are inaccessible.

- The only way to access a local variable of a calling function from the function it calls is to use a reference parameter. This gives access to the memory position via another name, i.e. the name of the (formal) reference parameter.
- The memory position of a reference parameter is not destroyed when a function terminates. Only the name is no longer valid.

Programming principles

It is important always to initialise variables before trying to use their values.

Don't use the same names for actual and formal parameters (and any other variables in the program, for that matter).

Exercises

Exercise 23.1

Draw a series of variable diagrams for the program below.

```
1  // Exercise 23.1
2  #include <iostream>
3  using namespace std;
4
5  void test(int x, int & y, int z)
6  {
7     x += 10;
8     y += 10;
9     z += 10;
10 }
11
12 int main( )
13 {
14    int a = 6;
15    int b = 7;
16    int c = 8;
17    test(a, b, c);
18    return 0;
19 }
```

Exercise 23.2

Draw a series of variable diagrams for the program below.

```
1  // Exercise 23.2
2  #include <iostream>
3  using namespace std;
4
5  int a, b, c;
6
7  void funcP(int & i, int j)
8  {
```

```
    j++;
    i += j - 1;
    c--;
}

void funcQ(int & i, int j)
{
    int c;
    j++;
    c = j + 2;
    i += j + c;
}

int main( )
{
    a = 1;
    b = 2;
    c = 3;
    funcP(a, b);
    funcQ(b, c);
    return 0;
}
```

Exercise 23.3

Draw a series of variable diagrams for the program below.

```
// Exercise 23.3
#include <iostream>
using namespace std;

void funcC(int one, int two)
{
    one += 1;
    two += 2;
}

void funcB(int & one, int & two)
{
    int three;
    three = one;
    one = two;
    two = three;
    funcC(one, two);
}

void funcA(int & one, int & two)
{
    funcB(one, two);
}

int main( )
{
```

```
27      int x = 1;
28      int y = 2;
29      funcA(x, y);
30      return 0;
31  }
```

Part IV

Data structures

In Part III, we saw a way of organising code into functions. The principle that we used was to keep together what belonged together. Similarly, when we have to deal with a lot of data, we need a way to organise it. The principle is the same: keep together what belongs together.

A data structure is a collection of related data values stored under a single name and organised so that the individual values can be accessed separately.

Arrays are data structures for storing related data values of the same type. Structs are data structures for storing related data values of different types.

Lesson 24

One-dimensional arrays

Purpose of this lesson

In this lesson we look at our first data structure, namely arrays. In general, we use an array if we have many data values of the same type and that represent the same sort of thing. For example, say you have a list of the names of your friends. These are all of the same data type, namely string, and each of the strings represents the same sort of thing, namely a name of one of your friends. An array is an ideal way of storing all the names.

Activity 24.a

Write a program to do the following:

1. Input a list of exam results (between 1 and 100 results).
2. Calculate the average of the results.
3. Display a list of results greater than the average.

Test yourself

This program uses a one-dimensional array for storing the exam results. If you can write a program that uses an array correctly, compare your solution with ours in the activity solution. If you are happy that your solution is as good (or better) than ours, continue with the exercises at the end of this lesson. If not, work through the following subactivities and then try writing the program again.

Subactivity 24.a.i

Write a program that inputs 10 exam results and calculates their average. Note: It is not necessary to use an array for this question. Neither is it necessary to use ten separate variables. You should be able to do it with one or two variables and a `for` loop.

Subactivity solution

```
// Average of ten marks
#include <iostream>
```

```
using namespace std;

int main( )
{
   int mark, total;
   float average;

   total = 0;
   for (int i = 1; i <= 10; i++)
   {
      cout << "Enter a mark: ";
      cin >> mark;
      total += mark;
   }

   average = float(total) / 10;

   cout << "The average is " << average << endl;
   return 0;
}
```

The sum of the results is calculated as they are input, therefore it is not necessary to store the results in ten separate variables and then to calculate the sum. In our solution we store the results one after the other in the variable `mark` and add it to `total`, representing the total so far. Once we have input all the values, we calculate the average. Note how we type-cast `total` to a floating-point value to ensure that floating point division is performed instead of integer division.

Discussion

Say we had to display all the results on the screen after calculating the average. Then the trick used in the above solution wouldn't work at all. We'd have to store each of the values in separate variables to be able to display them again. This is where an array would come in handy. Instead of declaring 10 separate variables, we declare a single array with 10 elements.

Subactivity 24.a.ii

Explain in a sentence (write it down in your study notebook) what the following program does:

```
#include <iostream>
using namespace std;

int main( )
{
   int examMarks[25];

   for (int i = 0; i < 25; i++)
   {
      cout << "Enter a mark: ";
      cin >> examMarks[i];
```

```
   }

   cout << "The marks are: " << endl;
   for (int i = 0; i < 25; i++)
      cout << examMarks[i] << endl;

   return 0;
}
```

Subactivity solution

The program inputs 25 exam marks into an array (prompting the user for each value) and then displays the contents of the array on the screen (each element on a separate line).

Discussion

As you should be able to see from this program, an array is a single entity in which more than one similar data element can be stored. One name is used to refer to a whole set of related data elements. For example, if we want to use an array, say `examMarks`, to store 10 integers, we declare it as follows:

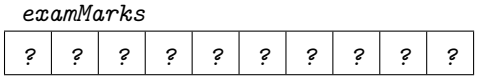
```
int examMarks[10];
```

In fact, it is even better to declare a constant first and then declare the array using the constant to specify the size. In other words

```
const int NUM_MARKS = 10;
int examMarks[NUM_MARKS];
```

The square brackets indicate that we want an array of integers (not just a single integer). The number between the brackets indicates how many integers we want, in this case 10.

A variable diagram for this array will look as follows:



`examMarks` represents ten values (or memory locations) that can be referred to as a whole, namely `examMarks`, or separately, namely as `examMarks[0]`, `examMarks[1]`, `examMarks[2]`, ... , `examMarks[9]`, each of which can store an integer value.

Note that `examMarks[0]` is the first element of the array and `examMarks[9]` is the last element. The number in square brackets (following the name of the array) is called the *subscript* (also sometimes referred to as an *index*). The *range* of the subscript in this instance is 0 to 9, which gives 10 values. A program using this array may include statements such as:

```
examMarks[5] = 50;
cin >> examMarks[0];
```

In the first statement, 50 is assigned to the sixth element of the array. The second statement inputs an integer and stores it in the first element of the array.

An important characteristic of arrays is that a variable can be used as a subscript. In other words, if `j` is an integer variable which assumes values from 0 to 9, we can use it to refer to the different elements of the array. For example:

```
j = 4;
examMarks[j] = 50;
j = 9;
cout << examMarks[j] << endl;
```

These statements will store the value 50 in the fifth element of `examMarks` and then output the tenth element.

Subactivity 24.a.iii

Write a program that inputs 25 exam marks, calculates their average and displays a list of the marks greater than the average.

Hint: The marks must be input and stored in an array. The total must be calculated as the marks are input. Once the average has been calculated, each element in the array has to be compared with it; this can be done with a `for` loop.

Subactivity solution

```
//Input 25 marks and display those greater than the average
#include <iostream>
using namespace std;

int main( )
{
   const int NUM_MARKS = 25;
   int examMarks[NUM_MARKS];
   int total;
   float average;

   total = 0;
   for (int i = 0; i < NUM_MARKS; i++)
   {
      cout << "Enter a mark: ";
      cin >> examMarks[i];
      total += examMarks[i];
   }

   average = float(total) / NUM_MARKS;
```

```
   cout << "The average is " << average << endl;

   cout << "The following marks were above average:" << endl;
   for (int i = 0; i < NUM_MARKS; i++)
      if (examMarks[i] > average)
         cout << examMarks[i] << endl;

   return 0;
}
```

Discussion

Compare the question in Subactivity 24.a.iii with the question in the main activity. There is one important difference between the two. The subactivity specifies that 25 results must be input, while the main activity does not specify how many: it can be anything from 1 to 100. How can a program handle this situation?

The problem is that the exact number of elements in an array must be specified when it is declared. The programmer has to find out what the maximum number of elements is and declare the array accordingly. The activity specifies a maximum of 100 elements and the array should therefore be declared accordingly.

As far as the rest of the program is concerned, there is one further complication: How does the program know when the last value has been entered? And what happens to the unused elements in the array?

One way to know when to stop requesting values is to first ask the user how many numbers are in the list and then adapt the `for` loop accordingly. Another way is to use a `while` loop which ends as soon as the last value has been entered; we shall of course have to identify the last value.

Subactivity 24.a.iv

Suppose any number of marks between 1 and 100 may be entered. Change the solution of Subactivity 24.a.iii so that the program first asks the user how many values there are, and then inputs the values and does the necessary calculations.

Subactivity solution

```
// Input a specified number of marks and
// display those greater than the average
#include <iostream>
using namespace std;

int main( )
{
   const int MAX_NUM = 100;
   int examMarks[MAX_NUM];
   int numMarks;
   int total;
   float average;

   cout << "How many marks will be entered? ";
```

```
   cin >> numMarks;
   while (numMarks < 1 || numMarks > MAX_NUM)
   {
      cout << "**Must be between 1 and " << MAX_NUM << "**" << endl;
      cout << "How many marks will be entered? ";
      cin >> numMarks;
   }

   total = 0;
   for (int i = 0; i < numMarks; i++)
   {
      cout << "Enter a mark: ";
      cin >> examMarks[i];
      total += examMarks[i];
   }

   average = float(total) / numMarks;

   cout << "The average is " << average << endl;

   cout << "The following marks were above average:" << endl;
   for (int i = 0; i < numMarks; i++)
      if (examMarks[i] > average)
         cout << examMarks[i] << endl;

   return 0;
}
```

Discussion

Note how this program starts with a `while` loop that keeps asking the user how many marks will be entered until a valid value (between 1 and 100) is provided. The reason why we do this data validation is that an invalid value would cause the subscript of the array to go outside its valid range (i.e. from 0 to 99) and this could lead to strange results.

The rest of the program changed very little. The two `for` loops are executed `numMarks` times and the average is calculated by dividing the total by `numMarks`.

Suppose the value of `numMarks` is 10, what happens to the other 90 elements in the array? The memory space occupied by these elements will be unused, and therefore wasted. This is one disadvantage of the way we have written the program.

The values of unused elements are unpredictable and a programmer must take the necessary precautions in a program not to index them. For example, if `i` in the last `for` loop of the program above ran from 0 to 99 when `numMarks` had a value of 10, we would get unpredictable results. (Try changing the last `for` loop in your program and test it with different values for `numMarks`. What happens?) Even though we do not assign values to all the elements, it is possible that there may be values in the memory locations assigned to the array by the declaration.

So always make sure that your programs only work with the elements of an array containing meaningful values.

Subactivity 24.a.v

Adapt the program you wrote for Subactivity 24.a.iv so that the program doesn't ask the user how many values there will be in advance. After each value, the program must ask the user whether there is another value. (The maximum number of values is 100.)

Subactivity solution

```
// Input an unspecified number of marks and
// display those greater than the average
#include <iostream>
using namespace std;

int main( )
{
   const int MAX_NUM = 100;
   int examMarks[MAX_NUM];
   int numMarks;
   int total;
   float average;
   char response;
   int i;

   total = 0;
   i = 0;
   do
   {
      cout << "Enter a mark: ";
      cin >> examMarks[i];
      total += examMarks[i];
      i++;
      cout << "More marks (Y/N): ";
      cin >> response;
   } while ((response == 'y' || response == 'Y') && i < MAX_NUM);

   numMarks = i;

   average = float(total) / numMarks;
   cout << "The average is " << average << endl;

   cout << "The following marks were above average:" << endl;
   for (int i = 0; i < numMarks; i++)
      if (examMarks[i] > average)
         cout << examMarks[i] << endl;

   return 0;
}
```

We use a `do..while` loop because the program must input at least one value before it can ask whether the user wants to enter another.

Discussion

In this program we still use `i` to refer to the elements of the array. Since a `do..while` loop does not increment a counter automatically, we have to increment the value of `i` inside the loop. Note how we initialize `i` to 0 before we enter the loop. (If we initialized it to 1 before the loop, no value would be stored in `examMarks[0]`.)

The rest of the statements in the loop ask for, and input the results. After each result, the program asks the user whether there are more results to be input, and if not, the loop will end. After incrementing, `i`'s value will be equal to the number of marks that have been input. So after the loop, we assign `i`'s value to `numMarks` and leave the rest of the program unchanged.

However, we are still not entirely happy with this program. I am sure that you would agree that it is quite irritating to have to type `Y` after each value when the above program is executed. There is another way of terminating an input list, and that is to use a *sentinel value*. (See Lesson 14 if you can't remember what a sentinel value is or how to use one to terminate input.)

Subactivity 24.a.vi

Rewrite the program of Subactivity 24.a.v to use a sentinel value of -1 to indicate the end of input. (This is the final version of this program and serves as the solution to the main activity.)

Activity solution

```
// Input an unspecified number of marks and
// display those greater than the average
#include <iostream>
using namespace std;

int main( )
{
   const int MAX_NUM = 100;
   int examMarks[MAX_NUM];
   int mark;
   int numMarks;
   int total;
   float average;
   char response;
   int i;

   total = 0;
   i = 0;
   do
   {
      cout << "Enter a mark (-1 to end): ";
      cin >> mark;
      if (mark != -1)
      {
```

```
            examMarks[i] = mark;

            total += examMarks[i];
            i++;
         }
      } while (mark != -1 && i < MAX_NUM);

      numMarks = i;

      average = float(total) / numMarks;
      cout << "The average is " << average << endl;

      cout << "The following marks were above average:" << endl;
      for (int i = 0; i < numMarks; i++)
         if (examMarks[i] > average)
            cout << examMarks[i] << endl;

      return 0;
}
```

Note how we use an `if` statement in the `do..while` loop to prevent -1 from being stored in the array and from being included in the calculation of the total. That's why we declared a separate variable `mark` to input a mark and test it before storing it in the array.

Activity 24.b

Write a program that generates 15 random numbers (between 0 and 999) and stores them in sorted order (i.e. in ascending order) in an array.

> **Test yourself**
>
> For this program you'll need an array of 15 places and you'll need to use the `rand` function. (The `rand` function was introduced in Lesson 17.)
>
> If you could write this program, compare your solution with ours. Otherwise work through the subactivities that follow.

Subactivity 24.b.i

Write a program to generate 15 random numbers (between 0 and 999) and store them (as they are generated) in an array.

Subactivity solution

```
// Generate 15 random integers and store them in an array
#include <iostream>
using namespace std;

int main( )
```

```
{
   const int NUM_VALS = 15;
   int values[NUM_VALS];
   int nextVal;

   srand(time(0));
   for (int i = 0; i < NUM_VALS; i++)
   {
      nextVal = rand( ) % 1000;
      values[i] = nextVal;
   }

   cout << "The values are:" << endl;
   for (int i = 0; i < NUM_VALS; i++)
      cout << values[i] << endl;

   return 0;
}
```

Discussion

There are (at least) two valid criticisms of this solution:

- We could replace the two statements in the body of the first `for` loop by a single statement, namely `values[i] = rand( ) % 1000;` and hence do without variable `nextVal`.
- We could display the values as we generate them. In other words, we could use a single `for` loop. In fact, we could even get by without an array at all!

However, we have done it like this to make things easier to adapt the program for the main activity. If the program you wrote included these or other short cuts, well done! We suggest however that you change it to conform to our method.

Subactivity 24.b.ii

Our solution to the previous subactivity is half-way to the final program. All we need to do is to insert each value in its correct place, rather than in the next place in the array.

What we want is that, at any point in the repetition of the first `for` loop, the values in the array thus far must be in sorted order. Say we are halfway through the loop repetitions, the array should look something like this:

values

14	*273*	*297*	*297*	*509*	*777*	*995*	*?*	*?*	*?*	*?*	*?*	*?*	*?*	*?*

Say, for example, that the next random number generated is 323. In your study notebook, write down what would need to be done to insert the new value in its correct place.

Subactivity solution

323 must obviously be inserted between 297 and 509. To insert it, we will have to move 509, 777 and 995 up to make place for it.

In general, to insert any new value, start at the beginning of the array and compare the new value with each value in turn until the new value is less than the current value, or the end of the list so far is reached.

If the end of the list has been reached, store the new value directly in the array. Otherwise, shift up all the rest of the values in the array by one place and then store the new value in the array.

That's simple enough to think about, but how do we code it in C++?

Subactivity 24.b.iii

What kinds of loops will be required to perform the above algorithm? Give reasons for your choice.

Subactivity solution

We need a while loop to find the position where the new value must be inserted, and a for loop to shift all the remaining values up one place.

The reasons for these choices are:

- *We use a while loop because we can't determine beforehand how many repetitions are needed before we find the position we are looking for. (We could use a do..while as well, but it is possible that we don't need any repetitions, i.e. if the new value must be placed in the first position.)*
- *We use a for loop because we can determine the number of repetitions that are needed to shift the rest of the elements up. The loop must run from the current position (i.e. the place where the new value must be inserted) up to the last value in the array thus far.*

Subactivity 24.b.iv

Write the two loops.

Subactivity solution

```
current = 0;
while (current < i && nextVal >= values[current])
   current++;
```

and

```
for (int j = i; j > current; j--)
   values[j] = values[j-1];
```

Note in the `while` loop how we use `i` as the number of the next open position. Note also how we use `current` as the loop control variable: We initialize it to 0 before the loop (to ensure that we start comparing `nextVal` with the value in position 0 of the array), we test it in the condition of the loop, and we change its value in the body of the loop. If `i` is 0, the loop is not executed.

Subactivity 24.b.v

Insert the two loops into the first loop of the program (in the solution to Subactivity 24.b.i) to obtain the solution to the main activity.

Activity solution

```
// Sort an array of 15 random integers
#include <iostream>
using namespace std;

int main( )
{
   const int NUM_VALS = 15;
   int values[NUM_VALS];
   int nextVal;
   int current;

   srand(time(0));
   for (int i = 0; i < NUM_VALS; i++)
   {
      nextVal = rand( ) % 1000;
      current = 0;
      while (current < i && nextVal >= values[current])
         current++;
      for (int j = i; j > current; j--)
         values[j] = values[j-1];
      values[current] = nextVal;
   }

   cout << "The values are:" << endl;
   for (int i = 0; i < NUM_VALS; i++)
      cout << values[i] << endl;

   return 0;
}
```

Note in this program that we do not have an `if` statement to distinguish between the situation where we want to insert a value in the middle of the list, or just add it to the end of the list. The nested `for` loop (namely `for (int j = ...)`) seems to be executed in either case. However, if the new value must be added at the end, `current` will be equal to `i` (for example, consider inserting the first value in the array) and the body of the `for` loop will not be executed at all.

Activity 24.c

Snoepie's Spaza Shop sells snacks and sweets. Apart from the problem of having to remember the prices of the different items, keying in each price when adding up the total is tedious.

Write a program so that Snoepie can enter a code for each item, see it's price on the screen, and see the total of the purchases. Here is an example of a test run:

```
Enter the item codes, -1 to end:
7
        R2.65
7
        R2.65
13
        R4.50
7
        R2.65
0
        R4.90
-1
Total:  R17.35
```

Note that the prices of items should be built into the program. They should not be entered as part of the input.

Test yourself

There are many ways of writing this program. You might have thought of a way already. For example, you could define a whole lot of constants for the prices of the items. This plan should bother you however, not just because this is a lesson about arrays, but because it would be difficult to use a code to determine which constant to use. If you used an array, you could store the prices in each position of the array corresponding to their codes.

Our purpose with this activity is to introduce an easy way of initialising an array. The first subactivity shows how this can be done.

Before we give the final solution to the activity, we deal with the important problem of the *index-out-of-range* error in some further subactivities.

Subactivity 24.c.i

Write a program that doubles all the values in an array called `a`. Use the following declaration:

```
int a[] = {10, 11, 12, 13, 14, 15};
```

The program should not do any input or produce any output.

Subactivity solution

```
//Double the values in an array

int main( )
{
   int a[] = {10, 11, 12, 13, 14, 15};
```

```
   for (int i = 0; i < 6; i++)
      a[i] *= 2;

   return 0;
}
```

(It may look funny, but since we do not do any input or output, we don't need the customary `#include <iostream>` and `using namespace std;` statements.)

Discussion

Although this is a somewhat stupid exercise, it illustrates a number of important things. Firstly, we see that we can initialise an array in its declaration. This is done by listing the values to be stored in the array in curly brackets. Note that this must be done in the declaration statement. You can't store a list of values in an array after its declaration. In other words, the following is illegal:

```
int a[6];
a = {10, 11, 12, 13, 14, 15};
```

Note also that the size of the array need not be specified when an array is initialised in its declaration. The size of the array (and the corresponding amount of memory that needs to be allocated) is determined from the number of elements in the list.

In fact, what we should have done was to declare a constant just before the array to store the number of elements in the array. Then wherever we require the size of the array in the program, we should use the constant. (Think how the above program should be fixed to incorporate this idea.)

Subactivity 24.c.ii

Try to write the program for the main activity now. Declare and initialise an array of floating point values. You can use any values you like. We used the following:

```
4.90 5.45 5.45 7.95 1.10 1.10 1.85 2.65
2.65 2.65 1.85 10.35 4.50 4.50 11.25 6.20
```

Subactivity solution

```
//Snoepie's Spaza Shop
#include <iostream>
using namespace std;

int main( )
{
   float prices[] = {4.90, 5.45, 5.45, 7.95,
                     1.10, 1.10, 1.85, 2.65,
```

```
                    2.65, 2.65, 1.85, 10.35,
                    4.50, 4.50, 11.25, 6.20};
   int code;
   float total = 0.00;

   cout << "Enter the item codes, -1 to end:" << endl;
   cin >> code;

   cout.setf(ios::fixed);
   cout.precision(2);
   while (code != -1)
   {
      cout << "\tR" << prices[code] << endl;
      total += prices[code];
      cin >> code;
   }
   cout << "Total: R" << total << endl;

   return 0;
}
```

We hope you are impressed with how simple and powerful the above program is!

There are two things to note about it. Firstly, by storing the prices in the array in positions corresponding to their codes, we can use variable code to index the prices directly. Secondly, we didn't declare a constant specifying the size of the array as we said we should in Subactivity 24.c.i. This was because we didn't need the size anywhere in the program. As we will see in the next subactivity, this was a mistake. As a general rule, it is a good idea to always declare a constant with the size of the array when you declare an array (whether you initialise the array in declaration or not).

Subactivity 24.c.iii

If you were naughty and didn't attempt the previous subactivity, but just looked at our solution, do yourself a favour and write the program yourself now. Don't look at our solution again. See if you can remember what we did and do it again yourself. (Your program need not look exactly like ours. You might like to use other variable names, or make some other improvements.)

When you have got the program compiling and running, test it out with some invalid codes, for example 27 and -5. Explain the result.

Subactivity solution

When we ran the program, we got a price of R0.00 for item 27 and R7834150034052657700000000000000000.00 for item -5!

The problem is that the program refers to positions outside the range of the array. In position 27 (which is 12 positions after the last element of the array) the value 0.00 was stored. In position -5 (which is 5 positions before the first element of the array) a lot of garbage was stored.

Note that the compiler didn't complain, neither did the program crash. The program used the garbage values as if they were correct.

The moral of the story is that you must always be very careful not to allow an index to go out of range when using an array.

Subactivity 24.c.iv

Fix the program of Subactivity 24.c.ii so that it prevents index-out-of-range errors from occurring. In other words, any codes less than -1 or greater than the greatest index in the array must be rejected with an error message.

We suggest you write a function to input a code and ensure that it is valid.

Activity solution

```
//Snoepie's Spaza Shop
#include <iostream>
using namespace std;

void getCode(int & codeP, int maxCode)
{
   cin >> codeP;
   while (codeP < -1 || codeP > maxCode)
   {
      cout << "Invalid code. Please re-enter" << endl;
      cin >> codeP;
   }
}

int main( )
{
   const int NUM_ITEMS = 16;
   float prices[] = {4.90, 5.45, 5.45, 7.95,
                     1.10, 1.10, 1.85, 2.65,
                     2.65, 2.65, 1.85, 10.35,
                     4.50, 4.50, 11.25, 6.20};
   int code;
   float total = 0.00;

   cout << "Enter the item codes, -1 to end:" << endl;
   getCode(code, NUM_ITEMS-1);

   cout.setf(ios::fixed);
   cout.precision(2);
   while (code != -1)
   {
      cout << "\tR" << prices[code] << endl;
      total += prices[code];
      getCode(code, NUM_ITEMS-1);
   }
   cout << "Total: R" << total << endl;

   return 0;
}
```

You might have designed your function differently. That's fine, as long as you sent the size of the array or the maximum value as a parameter.

In general, you should always make sure that there is no way that your program could refer to values outside the range of the array, i.e. make an index-out-of-range error. An index-out-of-range error is particularly dangerous because the program will compile correctly and run without crashing. It appears as if everything is hunky-dory, but the results will very often be incorrect. Even worse, the results might be correct sometimes, and incorrect others, which makes it very difficult to realise that there is an error, and to find and fix the mistake when it does happen.

In general, you should specify the size of the array explicitly, preferably with an integer constant, just before the declaration of the array. Use this constant to check that no indices go out of range. It is better to declare the size as a constant than use a literal value in the code where you test for this error. In this way, if you want to change the size of the array in future, you can just change the constant value, and you need not search through the whole program, checking for places where the size is used and changing them.

For example, say Snoepie wants to add a number of items to his selection, he can just add the new prices to the list in the array declaration and adjust `NUM_ITEMS` accordingly. The program will still work correctly.

Important points in this lesson

Programming concepts

In this lesson we saw how a set of values of the same type can be stored in a data structure called an array.

An array is declared as follows:

```
Type ArrayName[NumValues];
```

The square brackets indicates that *`ArrayName`* is an array. *`Type`* indicates the type of values to be stored in the array. *`NumValues`* indicates the number of values that can be stored in the array.

We can also initialise an array during declaration:

```
Type ArrayName[] = ListOfValues;
```

`ListOfValues` is a list of *`Type`* literals, separated by commas. When initialising an array in its declaration, *`NumValues`* must correspond with the number of values in *`ListOfValues`*.

We refer to specific elements in the array with, for example, *`ArrayName`*`[`*`Index`*`]`, where *`Index`* may assume any value in the range from 0 to 1 less than the number of values in the array.

As an alternative to initialising an array in its declaration, we can use a `for` loop to input values into the array. For example, if array `a`'s size is 10, we can use the following `for` loop to input its values:

```
for (int i = 0; i < 10; i++)
{
   cout << "Enter a value: ";
   cin >> a[i];
}
```

Similarly we can use a `for` loop to display the values of an array:

```
for (int i = 0; i < 10; i++)
   cout << a[i] << endl;
```

Note that it is not possible to input the contents of an entire array from the keyboard with a single statement like `cin >> a;`. Neither can we display all the elements of an array with a statement like `cout << a;`. To input or output the values of an array, we have to do so one element at a time. It is also not possible to assign the contents of an entire array to another similar array, e.g. `a = b;`. This has to be done one element at a time.

Programming principles

An array is used when a group of related data elements must be stored for later use in the program. Sometimes, even when a program needs to work with a lot of different data values, an array is necessary. For example, say a program must calculate the sum of 20 values. It is not necessary to store the values in an array; the sum can be calculated as the values are input. There is no need to have the values available later on. On the other hand, if all twenty values are required later on, it will be necessary to have access to all the values that were input, and an array will be needed.

Rather declare a constant representing the number of values to be store in the array than specify it with a literal value.

```
const int NUM_VALS = NumValues;
Type ArrayName[NUM_VALS];
```

This makes it easier to change the number of values to be stored in the array if needed, and makes it easier to see what is being done when the number of values is used in other parts of the program, e.g. in controlling the number of repetitions of a loop.

If the values that need to be stored in an array are known before the program starts, rather initialise the array in its declaration than input them via the keyboard.

If the exact number of values that will be stored in an array cannot be determined, declare the array to make provision for the maximum number of elements. Then use a `while` or `do..while` loop to input and store the values in the array. You'll also have to decide how to determine when all the values have been input. One way is to ask the user repeatedly whether there are more values. Another way is to include a sentinel in the input values to signal the end of the input.

If certain elements of an array are unused (i.e. uninitialised), you (i.e. the programmer) must make sure that these elements are never referred to in the program.

Also, beware of allowing an index to go out of range. The program will work with memory positions with unpredictable values, and which can change unpredictably during execution of the program.

Exercises

Exercise 24.1

Write a program that finds the greatest element in an array and swaps this element with the first element of the array. Display the array before and after it has been changed.

Note, the program should not input the values. The array should be initialised with the following values:

10	3	56	7	0	5	44	99	76	1

After swapping the greatest element, it should look as follows:

99	3	56	7	0	5	44	10	76	1

Exercise 24.2

Write a program that inputs 10 floating point numbers and displays them in reverse order.

Exercise 24.3

Write a program that determines whether the values in an array are stored in ascending order. The array should contain 20 elements, and the program should initialise it in its declaration. The program should display an appropriate message.

Exercise 24.4

Write a program that inputs 10 floating-point numbers into an array and then assigns these values to a second array of the same type.

Exercise 24.5

If you think about it, there's another way to find the correct position for inserting a value in an array to keep it sorted: Start at the end of the values stored thus far rather than at the beginning of the array. The algorithm for this would be:

> To insert any new value, start at the last value in the array so far. If the new value is greater than the last value, store the new value in the next position of the array. Otherwise compare the new value with each value in turn (moving backwards) until the new value is greater than the current value, or the beginning of the array is reached. Shift each value one place up as you search for the correct position. Then you can insert the value as soon as you find its position.

The advantage of this algorithm is that it only requires a single (`while`) loop to find the correct position and simultaneously shift up values to make place for the new value.

Adapt the solution to Activity 24.b to implement this algorithm.

Lesson 25

Arrays as parameters

Purpose of this lesson

The purpose of this lesson is to see how arrays can be transferred as parameters between the `main` function and other functions.

Activity 25.a

Adapt the program that we wrote for Activity 22.a so that it displays a table of all the data, the subtotals and grand total. To save you paging back to that lesson, here is an updated version of the problem:

> Patricia sells a variety of crafty items at the local flea market. In order to keep track of her income and expenditure, she wants a program that can add up the total value of her stock.

Write a program that prompts the user with the name of each item and its price, and inputs the amount of stock of that item. It should then display a table with these values, as well as an additional column for the stock value of each item. The program should also calculate and display the total stock value.

Use the following declarations of arrays to store the data:

```
const int NUM_ITEMS = 7;
string names[] = {"Plain candles", "Scented candles", "Dream-catchers", "Bead place mats",
                                   "Bead coasters", "Incense holders", "Hand-painted Ms"};
float prices[] = {8.00, 10.00, 19.00, 11.50, 5.00, 6.40, 23.50};
int stock[NUM_ITEMS];
```

Test yourself

Since the program must display a table of all the input, all this data will have to be stored. Arrays are ideal for this.

You could write this whole program without any additional functions. However, apart from the fact that this lesson is about sending arrays as parameters, the program will be quite unwieldy if all the code were placed in the main function. You should therefore use (at least two) functions to deal with different parts of the problem.

We suggest you use a function to input the stock numbers, and another function to calculate the results and output the table.

Subactivity 25.a.i

What is strange about the following program that uses functions with array parameters?

```
//Weird array parameters
#include <iostream>
using namespace std;

const int SIZE = 10;

void inputArray(int arrayP[])
{
   cout << "Enter " << SIZE << " values:" << endl;
   for (int i = 0; i < SIZE; i++)
      cin >> arrayP[i];
}

void outputArray(const int arrayP[])
{
   cout << "The values are" << endl;
   for (int i = 0; i < SIZE; i++)
      cout << arrayP[i] << ' ';
   cout << endl;
}

int main( )
{
   int array[SIZE];

   inputArray(array);
   outputArray(array);

   return 0;
}
```

In particular, compare how the array parameters used in this program differ from the reference and value parameters that we have used up to now for normal variables.

Subactivity solution

The array parameter of the `inputArray` function is strange for the following reasons:

- Even though the function is meant to change the values of the array, the `&` symbol is not used between the type and the name of the array parameter, as is normally done when a reference parameter is used.
- No size is specified in the square brackets after the name of the array. The size is specified as a global constant. (There is a good reason for this which we discuss below.)

The array parameter of the `outputArray` function is strange for the following reasons:

- The reserved word `const` is used before the array parameter.

- Like the `inputArray` function, no size is specified in the square brackets after the name of the array.

Discussion

Array parameters are different from the parameters we have seen up until now because all arrays are passed by reference by default. (Arrays cannot be passed by value because this would be inefficient. Pass-by-value would involve copying the entire array.)

The fact that all array parameters are passed by reference can be a problem however, because a function can inadvertently change the values in an array when it shouldn't. To prevent a function from changing the values of an array parameter, we use the `const` reserved word.

So, if you want to send an array as a reference parameter, leave out the `&`, and if you want to send it as a value parameter, use `const`. (Even though it isn't a proper value parameter, the const reserved word will prevent the function from changing any values in the array. The correct name for this type of parameter is a *constant reference* parameter.)

The main reason why the size of the array is not specified in the formal array parameter is that C++ ignores any size specified between the square brackets of the array parameter. In other words, even if we specified the array parameter with

```
void inputArray(int arrayP[SIZE])
```

C++ would allow us to send an array of any size (larger or smaller than `SIZE`).

The reason for this strange arrangement is so that a function can be called with arrays of different sizes. To do this, we have to include a separate parameter to specify the size. For example, with the header:

```
void inputArray(int arrayP[], int n)
```

the main function could declare two arrays of different sizes, say

```
int arrayA[20];
int arrayB[50];
```

and then call function `inputArray` with either of the arrays, namely

```
inputArray(arrayA, 20);
inputArray(arrayB, 50);
```

If the size of `arrayP` was fixed, we couldn't do this.

Sometimes we don't know how many values have been stored in the array until the program is running. Then we also use an additional parameter to tell the function how many elements to work with. We will see the use of this trick below.

Subactivity 25.a.ii

Rewrite the above program to use a local rather than a global constant. In other words, move the declaration of constant `SIZE` to the `main` function.

Subactivity solution

If you simply move the declaration, the program won't compile any more because both `inputArray` and `outputArray` refer to `SIZE`. You need to include an additional parameter in the function headers to pass the size of the array to them:

```
// Weird array parameters
#include <iostream>
using namespace std;

void inputArray(int arrayP[], int n)
{
   cout << "Enter " << n << " values:" << endl;
   for (int i = 0; i < n; i++)
      cin >> arrayP[i];
}

void outputArray(const int arrayP[], int n)
{
   cout << "The values are" << endl;
   for (int i = 0; i < n; i++)
      cout << arrayP[i] << ' ';
   cout << endl;
}

int main( )
{
   const int SIZE = 10;
   int array[SIZE];

   inputArray(array, SIZE);
   outputArray(array, SIZE);

   return 0;
}
```

This is the most common way of passing arrays as parameters. Note that if you send more than one array to a function, and the arrays all have the same size, then you only need to send one additional parameter for the size (not one for each array parameter).

Discussion

Here is a brief explanation of why C++ doesn't allow array parameters to be passed by value:

Suppose we had an array of 1000 floating point values. If this array could be sent to a function as a value parameter, a copy of the complete array would have to be made in the computer's memory for use by the function. If the amount of available memory was limited, this extra array would occupy valuable space. (Memory is very cheap these days, so this wouldn't be a real problem.) A more serious problem is that it would slow the execution of the program down, since it would take time to make the copy.

Remember that when a parameter is passed by reference, it is in fact just a temporary name for the actual parameter used in the function call, not a copy of the actual parameter. The same is true for array parameters. Since they are passed by reference, an array parameter is in fact exactly the same array as the actual parameter. It just has another (temporary) name. No additional memory space is involved, and no copying is required.

For this reason, C++ only allows arrays to be passed by reference even when the subprogram doesn't need to change the values of the array. To prevent a function from changing the values in an array, we use the `const` reserved word.

Subactivity 25.a.iii

Before attempting to write a program for the main activity, write descriptions and headers for the two functions that will be called from the `main` function.

Subactivity solution

For the first function

```
//Input the stock numbers of all items
void inputData(const string namesP[], const float pricesP[], int stockP[], int n)
```

and for the second function

```
//Calculate the stock value of each item and the total value, and output a table
void calcAndOutput(const string namesP[], const float pricesP[], const int stockP[], int n)
```

Note firstly the single additional parameter to specify the size of the arrays. Note also the use of the `const` reserved word for all arrays that will not be changed by the function.

Subactivity 25.a.iv

Write a skeleton of the program, including stubs of the two functions, and the main function containing the declaration of the arrays, as well as calls of the two functions. By a *function stub*, we mean the function name with its parameters, and a simple `cout` statement displaying the function's name in its body.

To save yourself typing, edit the solution to Activity 22.a.

Subactivity solution

```
//Stock totals
```

```
#include <iostream>
#include <string>
using namespace std;

//Input stock numbers of all the items
void inputData(const string namesP[], const float pricesP[], int stockP[], int n)
{
   cout << " This is inputData " << endl;
}

//Calculate the stock value of each item and the total
//stock value, and output a table
void calcAndOutput(const string namesP[], const float pricesP[], const int stockP[], int n)
{
   cout << " This is calcAndOutput " << endl;
}

int main( )
{
   const int NUM_ITEMS = 7; //must correspond with the number of names and prices
   string names[] = {"Plain candles", "Scented candles", "Dream-catchers", "Bead place mats",
                                      "Bead coasters", "Incense holders", "Hand-painted Ms"};
   float prices[] = {8.00, 10.00, 19.00, 11.50, 5.00, 6.40, 23.50};
   int stock[NUM_ITEMS];

   inputData(names, prices, stock, NUM_ITEMS);
   calcAndOutput(names, prices, stock, NUM_ITEMS);

   return 0;
}
```

You might like to compile and run the program at this stage to check that it is correct so far.

Subactivity 25.a.v

Write the `inputData` function so that it prompts the user for each of the stock numbers.

Subactivity solution

```
//Input stock numbers of all the items
void inputData(const string namesP[], const float pricesP[], int stockP[], int n)
{
   for (int i = 0; i < n; i++)
   {
      cout << namesP[i] << ": R" << pricesP[i] << endl;
      cout << "Stock? ";
      cin >> stockP[i];
   }
}
```

Subactivity 25.a.vi

Now write the function `calcAndOutput` to receive all the arrays as parameters and to calculate the stock value of each item as well as the total stock value, and to display a table of all this data.

Subactivity solution

```
//Calculate the stock value of each item and the total stock value, and output a table
void calcAndOutput(const string namesP[], const float pricesP[], const int stockP[], int n)
{
   float stockVal;
   float stockTotal = 0;

   cout << "Item name\tPrice\tStock\tStock value" << endl;
   cout << "---------\t-----\t-----\t-----------" << endl;
   for (int i = 0; i < n; i++)
   {
      stockVal = pricesP[i] * stockP[i];
      cout << namesP[i] << "\tR" << pricesP[i] << "\t" << stockP[i] << "\tR" << stockVal
                                                                              << endl;
      stockTotal += stockVal;
   }
   cout << "Total stock value: R" << stockTotal << endl;
}
```

Note how we can dispense with the function that we used in Activity 22.a to deal with each item since we are using an array. We can now use a loop to deal with the repetition.

Now use the solutions to Subactivities 25.a.v to 25.a.vi above to write the program asked for in the main activity.

Activity solution

```
//Stock totals
#include <iostream>
#include <string>
using namespace std;

//Input stock numbers of all the items
void inputData(const string namesP[], const float pricesP[], int stockP[], int n)
{
   for (int i = 0; i < n; i++)
   {
      cout << namesP[i] << ": R" << pricesP[i] << endl;
      cout << "Stock: ";
      cin >> stockP[i];
   }
}

//Calculate the stock value of each item and the total
//stock value, and output a table
```

```
void calcAndOutput(const string namesP[], const float pricesP[], const int stockP[], int n)
{
   float stockVal;
   float stockTotal = 0;

   cout << "Item name\tPrice\tStock\tStock value" << endl;
   cout << "---------\t-----\t-----\t-----------" << endl;
   for (int i = 0; i < n; i++)
   {
      stockVal = pricesP[i] * stockP[i];
      cout << namesP[i] << "\tR" << pricesP[i] << "\t" << stockP[i] << "\tR" << stockVal
                                                                              << endl;
      stockTotal += stockVal;
   }
   cout << "Total stock value: R" << stockTotal << endl;
}

int main( )
{
   const int NUM_ITEMS = 7; //must be correspond with the //number of names and prices
   string names[] = {"Plain candles", "Scented candles", "Dream-catchers", "Bead place mats",
                                      "Bead coasters", "Incense holders", "Hand-painted Ms"};
   float prices[] = {8.00, 10.00, 19.00, 11.50, 5.00, 6.40, 23.50};
   int stock[NUM_ITEMS];

   inputData(names, prices, stock, NUM_ITEMS);
   calcAndOutput(names, prices, stock, NUM_ITEMS);

   return 0;
}
```

Important points in this lesson

Programming concepts

This lesson focussed on the use of arrays as parameters in functions. The general form of a function that takes an array as parameter is

```
ReturnType FunctionName(Type ArrayName[])
{
   Statements;
}
```

There is no point in specifying the size of the array parameter in the square brackets, C++ simply ignores it. If the function requires the size of the array, it should either be specified as a global constant before the function definition, eg.

```
const int NUM_ELEMS = NumElems;
```

or the size can be sent as an additional parameter:

```
ReturnType FunctionName(Type ArrayName[], int NumElemsP)
```

```
{
   Statements;
}
```

In C++, all array parameters are passed by reference. This is to save copying the entire contents of an array as would have to be done if it were passed by value. The `&` operator is not required (and should not be used) with an array parameter.

To prevent a function from changing the values of an array parameter, the `const` reserved word has to be used before *`Type`* in the parameter list. This is called *pass by constant reference*.

Functions cannot return an array as the return type. In other words, we cannot use a value-returning function to return a whole array to a calling function.

Programming principles

The decision of whether the size of the array should be specified as a global constant or as a separate integer parameter depends on whether you want to allow the function to deal with arrays of different sizes, or restrict the number of elements that should be accessed.

Exercises

Exercise 25.1

Write a program that does the following:

- Inputs 10 integers into an array.
- Inputs an integer and reports whether it appears in the array or not.
- Determines what the smallest element in the array is and displays it.
- Reverses the order of the elements in the array (the first element will be in the last position, the second element in the penultimate position, etc.) and displays the resulting array.

Use functions as appropriate.

Exercise 25.2

Peter the postmaster got bored with his job and did an experiment with the 50 post boxes at Putsonderwater Post Office. Starting with post box 2, he opened the doors of all the post boxes with even numbers. Then, starting with post box 3, he went to every third post box and closed the door if it was open and opened it if it was closed. After that, starting with post box 4, he did the same with every fourth post box. Then with every fifth one and then with every sixth one, etc. He was quite surprised to see how the closed post boxes were eventually distributed. Write a modular program to simulate these activities. Which post boxes were eventually closed?

Hint: Use an array with 50 Boolean values to represent the post box doors. If an element is `false`, the corresponding door is closed.

Exercise 25.3

Students wrote a test consisting of 10 questions to which they had to reply T or F (for true or false). Each student's answers are encoded as a sequence of ten F and T characters, e.g. FTFFFTTFTF. The correct answers are also encoded as a sequence of F and T characters.

Write a program to input the sequence of correct answers and then the sequence of replies for one student. It must then calculate the student's mark out of ten and display the result.

Lesson 26

Two-dimensional arrays

Purpose of this lesson

Data must often be stored in table form. In Lessons 24 and 25, we worked with *one-dimensional* arrays which corresponded to data in list form. In this lesson we look at *two-dimensional* arrays that are used to store data in table form.

In the activities in this lesson you will see how to input data into a two-dimensional array, how to display the data in such an array and how to do calculations on this data. We start by explaining what a two-dimensional array is and how it is declared.

Activity 26.a

A motor manufacturer has collected information about the noise levels (in decibels) of five different motor cars. Each car's noise level has been measured at six different speeds. The data looks as follows:

	20km/h	40km/h	60km/h	80km/h	100km/h	120km/h
Car 1	90	94	102	111	122	134
Car 2	77	80	86	94	103	113
Car 3	83	85	94	100	111	121
Car 4	71	76	85	96	110	125
Car 5	84	91	98	105	112	119

Write a program that inputs this data and displays it in an appropriate format on the screen. It must then calculate the average noise level of each car.

Test yourself

In order to write this program, you have to know how to store data in a two-dimensional array, how to display the values and how to do calculations on the data. The following subactivities develop the program step-by-step and show (amongst other things) how to declare a two-dimensional array, and how to pass a two-dimensional array as a parameter.

Subactivity 26.a.i

Consider the following declarations that we will be using for this program:

```
const int NUM_CARS = 5;
const int NUM_SPEEDS = 6;
int noiseData[NUM_CARS][NUM_SPEEDS];
```

Using these declarations, write C++ statements in your study notebook to do each of the following:

(i) Store the value 90 in row 0, column 0 of the array.

(ii) Input a value for the 4th car at the fastest speed.

(iii) Display the value of the ith car at the jth speed.

Subactivity solution

(i) `noiseData[0][0] = 90;`

(ii) `cin >> noiseData[3][5];`

(iii) `cout << noiseData[i-1][j-1];`

Discussion

The table consists of five rows (one for each motor car) and six columns (one for each speed). The subscripts of two-dimensional arrays also start with 0, so in this array, the first subscript ranges from 0 to 4, and the second from 0 to 5. That's why we subtracted 1 from the row and column subscripts `i` and `j` in (iii) above.

But why declare the dimensions of the array as two constants? The problem statement specifies that there are 5 cars and 6 speeds. It is good programming practice to store this type of information in constants. We could declare the array with `int noiseData[5][6];` but this would make it difficult to change the program if the number of cars or speeds changed. For example, if the program needs to be used later for a similar application where there are 10 cars and 4 different speeds, all we will need to change is the values of these constants. The rest of the program will not ned to be changed at all.

Subactivity 26.a.ii

Write a program that inputs and stores data in the array declared in Subactivity 26.a.i. Do this all in the `main` function. Prompt the user as follows:

```
Enter 6 noise levels for car no 1
At 20km/h: 90
At 40km/h: 102
At 60km/h: 111
etc...
```

Hint: Use nested `for` loops that range from 0 to 1 less than `NUM_CARS` and 0 to 1 less than `NUM_SPEEDS`, respectively.

Use the data as set out in the table given in the main activity to test your program.

Subactivity solution

```
// Noise levels of cars
#include <iostream>
using namespace std;

int main( )
{
   const int NUM_CARS = 5;
   const int NUM_SPEEDS = 6;
   int noiseData[NUM_CARS][NUM_SPEEDS];
   int speed;

   for (int i = 0; i < NUM_CARS; i++)
   {
      cout << "Enter " << NUM_SPEEDS << " noise levels for car no " << i+1 << endl;
      for (int j = 0; j < NUM_SPEEDS; j++)
      {
         speed = 20 * (j+1);
         cout << "At " << speed << "km/h: ";
         cin >> noiseData[i][j];
      }
   }
   return 0;
}
```

Discussion

It is standard practice to use nested loops (normally `for` loops) to work through the values in a two-dimensional array. Note how the outer loop runs from 0 to 1 less than the number of rows, and the inner loop runs from 0 to 1 less than the number of columns. This means that all the values for one row are input before the values for the next row are input.

Subactivity 26.a.iii

Consider the following adaption of the above program that uses a function to input the data into the array:

```
// Noise levels of cars
#include <iostream>
using namespace std;

const int NUM_CARS = 5;
const int NUM_SPEEDS = 6;
```

```
void getData(int noiseDataP[][NUM_SPEEDS])
{
   int speed;
   for (int i = 0; i < NUM_CARS; i++)
   {
      cout << "Enter " << NUM_SPEEDS << " noise levels for car no " << i+1 << endl;
      for (int j = 0; j < NUM_SPEEDS; j++)
      {
         speed = 20 * (j+1);
         cout << "At " << speed << "km/h: ";
         cin >> noiseDataP[i][j];
      }
   }
}

int main( )
{
   int noiseData[NUM_CARS][NUM_SPEEDS];

   getData(noiseData);

   return 0;
}
```

What is strange about this program, particularly the way the two-dimensional array is specified as a parameter?

Subactivity solution

The first subscript of the array parameter is not specified (as we did for one-dimensional array parameters), but the second subscript is specified.

Also, we did not include the subscript sizes as additional parameters to the function.

Note finally that we moved the declarations of `NUM_CARS` and `NUM_SPEEDS` to the beginning of the program, before all the functions. In other words, we've used global constants instead of local constants.

Discussion

The most important thing you need to remember is that the second subscript of a two-dimensional array parameter has to be specified. (Apart from the fact that passing arrays to functions in C++ is somewhat strange, this is perhaps the most unsatisfactory part. There is a complicated reason why the size of the second subscript has to be specified, but we don't explain it here. You must just remember that you have to do it this way.)

The other differences are a result of this requirement. We have to declare global constants to make them available to the function. (Note that a global constant is acceptable, whereas a global variable is not. The problem with a global variable is that different functions can change its value willy-nilly, causing confusion. A global constant can't be changed by different functions because it is a constant.)

We can get by without additional parameters specifying the size of the subscripts because the global constants are available to the function.

Subactivity 26.a.iv

Expand the program of Subactivity 26.a.iii to display the data that has been input, in table format on the screen. Your table should look like the one given in the main activity, with all the row and column headings.

Write a function that can be added to the above program to do this.

Subactivity solution

We only give the code for function `showData`. The rest of the program will remain exactly as it is, except for a call to the function, namely

```
showData(noiseData);
```

which must be inserted in the `main` function.

```
void showData(const int noiseDataP[],[NUM_SPEEDS])
{
   int speed;

   //Display headings
   for (int j = 0; j < NUM_SPEEDS; j++)
   {
      speed = 20 * (j+1);
      cout << '\t' << speed << "km/h";
   }
   cout << endl;

   //Display the data row by row
   for (int i = 0; i < NUM_CARS; i++)
   {
      cout << "Car " << i+1 << ": ";
      for (int j = 0; j < NUM_SPEEDS; j++)
         cout << '\t' << noiseDataP[i][j];
      cout << endl;
   }
}
```

Discussion

Note firstly that we use the `const` reserved word to indicate that function `showData` will not (and cannot) change the values in its array parameter.

Once again, we use nested loops to display the elements of the two-dimensional array. The inner loop outputs all the elements in a single row. The outer loop is executed as many times as there are rows in the array. The general structure for displaying a two-dimensional array is therefore

```
for (int i = 0; i < NUM_ROWS; i++)
{
   for (int j = 0; j < NUM_COLS; j++)
      cout << ArrayName[i][j] << ' ';
   cout << endl;
}
```

When `i` is 0, `j`'s value will run from 0 to 1 less than the number of columns, and for each value of `j`, the value of *`ArrayName`*`[0][j]` will be displayed. The `cout << endl;` will advance the display by one line before the values in the next row are displayed.

These `for` loops therefore display the data row-by-row. However, if we were to swap the two `for` statements around like this:

```
for (int j = 0; j < NUM_COLS; j++)
{
   for (int i = 0; i < NUM_ROWS; i++)
      cout << ArrayName[i][j] << ' ';
   cout << endl;
}
```

the data would be displayed column-by-column.

In the program above, we display the values row-by-row. Can you work out what the output of the program would look like if we displayed it column-by-column?

Subactivity 26.a.v

All that needs to be done to complete the program for the main activity is to write a function that calculates and displays the average noise level for each car. The function receives the array with the car data as a parameter, calculates the sum of the noise levels and determines the average for each car.

Add such a function to your program to obtain the solution to the main activity.

Activity solution

```
//Noise levels of cars
#include <iostream>
using namespace std;

const int NUM_CARS = 5;
const int NUM_SPEEDS = 6;

void getData(int noiseDataP[][NUM_SPEEDS])
{
   int speed;
```

```
   for (int i = 0; i < NUM_CARS; i++)
   {
      cout << "Enter " << NUM_SPEEDS << " noise levels for car no " << i+1 << endl;
      for (int j = 0; j < NUM_SPEEDS; j++)
      {
         speed = 20 * (j+1);
         cout << "At " << speed << "km/h: ";
         cin >> noiseDataP[i][j];
      }
   }
}

void showData(const int noiseDataP[][NUM_SPEEDS])
{
   int speed;

   //Display headings
   for (int j = 0; j < NUM_SPEEDS; j++)
   {
      speed = 20 * (j+1);
      cout << '\t' << speed << "km/h";
   }
   cout << endl;

   //Display the data row by row
   for (int i = 0; i < NUM_CARS; i++)
   {
      cout << "Car " << i+1 << ": ";
      for (int j = 0; j < NUM_SPEEDS; j++)
         cout << '\t' << noiseDataP[i][j];
      cout << endl;
   }
}

//Calculate and display the average noise level for each car
void calcAverages(const int noiseDataP[][NUM_SPEEDS])
{
   int totalNoise;
   float average;

   //Calculate the sum and average row by row
   cout << "The average noise level for each car:" << endl;

   for (int i = 0; i < NUM_CARS; i++)
   {
      totalNoise = 0;
      cout << "Car no " << i+1 << ": ";
      for (int j = 0; j < NUM_SPEEDS; j++)
         totalNoise += noiseDataP[i][j];
      average = float(totalNoise) / NUM_SPEEDS;
      cout << average << endl;
   }
}
```

```
int main( )
{
   int noiseData[NUM_CARS][NUM_SPEEDS];

   getData(noiseData);
   showData(noiseData);
   calcAverages(noiseData);

   return 0;
}
```

Discussion

Note in function `calcAverages` how we use single variables to calculate the respective totals and averages. Since we don't need to keep these values for each car, they can be overwritten for each successive car.

Say we were required to add a function to calculate the average noise levels at each speed (i.e. the averages of the columns in the table). The outer `for` loop would then run from 0 to 1 less than `NUM_SPEEDS`, and the inner `for` loop would run from 0 to 1 less than `NUM_CARS`.

Activity 26.b

A two-dimensional array with the same number of rows and columns is sometimes called a *matrix* (especially in a mathematical context). Write a program that stores values in an 4×4 matrix (in other words, an array with 4 rows and 4 columns) and then transposes it. To *transpose* a matrix, all values in the rows and columns have to be swapped across the diagonal. For example, if we transpose the first matrix below, we get the second matrix:

1	2	1	2
3	4	3	4
5	6	5	6
7	8	7	8

$\Longrightarrow$

1	3	5	7
2	4	6	8
1	3	5	7
2	4	6	8

Note how the values in the diagonal are unchanged.

The program should output the matrix before and after transposing it.

Note that the values of the initial matrix should be built into the program, i.e. they should not be input from the keyboard.

Test yourself

The only concept that we haven't covered for you to be able to write this program is how to initialise a two-dimensional array. Of course, you could write a lot of assignment statements, but there is an easier way. You can initialise a two-dimensional array in its declaration as we did for one-dimensional arrays. The first subactivity shows how to do this.

Then try to write the program yourself. But don't cheat! Don't simply display the matrix as if it was transposed by swapping the row and column subscripts. You should write a separate function the takes a single matrix as parameter and swaps the necessary values in it to transpose it.

If you get stuck, find the subactivity that addresses your problem and see if you can continue on your own.

Subactivity 26.b.i

Type in a skeleton of the program, consisting of the `main` function containing the declarations for a 4×4 integer matrix, `mat`, that is initialised to the values given in the main activity. See if you can guess how this is done. Remember that a two-dimensional matrix is an array of which each element is another array.

Subactivity solution

```
//Transpose a matrix
#include <iostream>
using namespace std;

const int N = 4;

int main( )
{
   int mat[][N] = {{1,2,1,2}, {3,4,3,4}, {5,6,5,6}, {7,8,7,8}};

   return 0;
}
```

Note that we declare the constant `N` as a global constant because we are using a two-dimensional array and we are going to want to use it inside functions and in the array parameter sent to functions.

Note also how we leave out the size of the first subscript of a two-dimensional matrix but specify the size of the second. The size of the first is determined implicitly from the number of rows specified in the initialisation list, but C++ can't determine the size of the second from the initialisation list (don't ask why!) so you have to specify it. Strange but true.

Subactivity 26.b.ii

Write a void function `showMatrix` to display an N×N matrix on the screen.

Subactivity solution

```
//Display the contents of a matrix on the screen
void showMatrix(const int matP[][N])
{
   for (int i = 0; i < N; i++)
   {
      for (int j = 0; j < N; j++)
         cout << matP[i][j] << '\t';
      cout << endl;
   }
}
```

It would be a good idea to insert this function in the program and test it at this stage.

Subactivity 26.b.iii

Now write a void function `transpose` to transpose an N×N matrix.

Hint: Use the `swap` function. This is a C++ library function not mentioned in Appendix D. It swaps the values of two integer variables. For example, `swap(x, y);` will swap the values of variables `x` and `y`. Similarly, `swap(mat[i][j], mat[j][i]);` will swap two values in matrix `mat`, namely the value in row `i` column `j` and the value in row `j` column `i`.

Subactivity solution

If you wrote the following function, it's wrong!

```
//Transpose the matrix
void transpose(int matP[][N])
{
   for (int i = 0; i < N; i++)
      for (int j = 0; j < N; j++)
         swap(matP[i][j], matP[j][i]);
}
```

If you are not convinced that it is wrong, enter it into the program we have been developing (with the other function) and try it out. You will see that (it appears as if) it does nothing.

The problem is that it swaps pairs of values in the array and then swaps them back again. For example, at some stage `i` will be 0 and `j` will be 1. Then the values `matP[0][1]` and `matP[1][0]` will be swapped. A bit later, `i` will be 1 and `j` will be 0. Then the values `matP[1][0]` and `matP[0][1]` will be swapped. The previous work is undone!

Another thing that should bother you is that at different stages `i` will be equal to `j` and we will be swapping `matP[0][0]` with `matP[0][0]`, `matP[1][1]` with `matP[1][1]`, etc. These values are all on the diagonal of the matrix and need not be swapped.

So we need to adjust the boundaries of the loop(s) to do less swaps. In fact, the only subscripts we need to look at are those that are shaded in the following diagram.

1	2	1	2
3	4	3	4
5	6	5	6
7	8	7	8

These values must be swapped with the corresponding values below the diagonal.

Subactivity 26.b.iv

If your function looked like the one above, fix it, taking the information above into account. Otherwise compare your function with the one in the following solution.

Subactivity solution

```
//Transpose the matrix
void transpose(int matP[][N])
{
   for (int i = 0; i < N-1; i++)
      for (int j = i+1; j < N; j++)
         swap(matP[i][j], matP[j][i]);
}
```

The only differences between this function and the one given above are the final value of `i` and the starting value of `j`.

Put all these functions together into the program to form the solution of the main activity:

Activity solution

```
//Transpose a matrix
#include <iostream>
using namespace std;

const int N = 4;

//Display the contents of a matrix on the screen
void showMatrix(const int matP[][N])
{
   for (int i = 0; i < N; i++)
   {
      for (int j = 0; j < N; j++)
         cout << matP[i][j] << '\t';
      cout << endl;
   }
}

//Transpose the matrix
void transpose(int matP[][N])
{
```

```
   for (int i = 0; i < N-1; i++)
      for (int j = i+1; j < N; j++)
         swap(matP[i][j], matP[j][i]);
}

int main( )
{
   int mat[][N] = {{1,2,1,2}, {3,4,3,4}, {5,6,5,6}, {7,8,7,8}};

   cout << "Before transposing" << endl;
   showMatrix(mat);

   transpose(mat);

   cout << "After transposing" << endl;
   showMatrix(mat);

   return 0;
}
```

Important points in this lesson

Programming concepts

Data in table form can be stored in a two-dimensional array. A two-dimensional array can be viewed as a one-dimensional array of which the elements are also one-dimensional arrays.

The declaration of a two-dimensional array is like a one-dimensional array. The difference is that two subscripts must be used for a two-dimensional array. When declaring a two-dimensional array, the size of the row and column subscripts must be specified, e.g.

```
Type ArrayName[NUM_ROWS][NUM_COLS];
```

NUM_ROWS and *NUM_COLS* can be integer literals, constants or variables.

Two-dimensional arrays with the same number of rows and columns are often referred to as *matrices*. An N×N matrix is a two-dimensional array with N rows and N columns.

A two-dimensional array can be initialised in its declaration statement. Since a two-dimensional array is really a one-dimensional array where each element is another one-dimensional array, each row of the array is listed in braces (curly brackets) and they are all put in braces. For example, the following will declare and initalise a 2×3 array of integers:

```
int mat2x3[][3] = {{1, 2, 3}, {4, 5, 6}};
```

The number of rows need not be specified in the declaration (as above) but the number of values in each row (i.e. the number of columns) must be specified, even if it is obvious from the initialisation list.

When passing a two-dimensional array as a parameter to a function, you need not specify the size of the first index (the number of rows) but you MUST specify the size of the second index (the number of columns) between the relevant square brackets of the formal parameter. The standard technique is to declare global

constants specifying the number of rows and columns, and using these in the function. The format therefore looks like this:

```
const int NUM_ROWS = NumRows;
const int NUM_COLS = NumCols;
:
void FunctionName(Type ArrayName[][NUM_COLS])
   :
```

The function can then use *NUM_ROWS* and *NUM_COLS* to ensure that it only accesses values in valid indices of the array.

If you want to allow the function to deal with arrays of different sizes, you can send an additional integer parameter specifying the number of rows:

```
const int NUM_COLS = NumCols;
:
void FunctionName(Type ArrayName[][NUM_COLS], int NumRows)
   :
```

Note that this only allows the function to deal with arrays with different numbers of rows. You CANNOT get a function to deal with arrays of different numbers of columns, since the number of columns MUST be specified in the square brackets for the second index as a constant or literal.

The above trick (of sending an additional parameter) is also useful for getting a function to only look at a restricted number of rows of an array. In other words, by providing a value for *NumRows* which is less than *NUM_ROWS*, we can get the function to only access some of the rows of the array.

Programming principles

We generally specify the size of the indices (i.e. the number of rows and columns) of a two-dimensional matrix as constants (like *NUM_ROWS* and *NUM_COLS*). This makes it much easier to change the size of the array if we ever need to in the future.

To step through the values of a two-dimensional array, we use two loops û one nested inside the other. To step through them row-by-row, the outer loop determines the row subscript and the inner loop determines the column subscript. For example:

```
for (int i = 0; i < NUM_ROWS; i++)
{
   for (int j = 0; j < NUM_COLS; j++)
      cout << ArrayName[i][j] << ' ';
   cout << endl;
}
```

To step through the values column-by-column, the loops will change as follows:

```
for (int j = 0; j < NUM_COLS; i++)
{
   for (int i = 0; i < NUM_ROWS; j++)
      cout << ArrayName[i][j] << ' ';
   cout << endl;
}
```

Exercises

Exercise 26.1

Write a program that will input two N×N matrices and determine whether one is the "transposition" of the other.

Exercise 26.2

Write a program that stores rainfall measurements for each month of the years 2000 to 2005 in a two-dimensional array. The program must then use this data to calculate and display the following:

- The average annual rainfall over the period 2000 to 2005.
- The average monthly rainfall over the period 2000 to 2005.
- The month from January 2000 to December 2005 with the highest rainfall.

Exercise 26.3

Write a program that does reservations for a theatre with 5 rows of seats, and 9 seats per row. The rows are numbered from 'A' to 'E' (back to front) and the seats from 1 to 9 in each row. The user should be presented with a seat plan (showing the current bookings) from which a number of seats in one row can be booked. For example

```
   1 2 3 4 5 6 7 8 9
A: - - - - - - - - -
B: - - - - - - - - -
C: - - - - - - - - -
D: - - - - - - - - -
E: - - - - - - - - -
     S C R E E N
In which row do you want seats? A
How many seats? 2
Starting at which number? 4
2 seats reserved in row A
   1 2 3 4 5 6 7 8 9
A: - - - R R - - - -
B: - - - - - - - - -
C: - - - - - - - - -
D: - - - - - - - - -
E: - - - - - - - - -
     S C R E E N
In which row do you want seats? A
How many seats? 5
Starting at which number? 5
PROBLEM: A5 is already reserved
In which row do you want seats? Q
```

Store the data in a two-dimensional array of characters. Initially all the elements of the array contain the character `-`, to indicate that the seats are available. When a reservation is made, the program must determine if the requested seats are available and if they are, it must reserve them. A reservation is made by assigning the character `R` to the elements in question. If any of the requested seats is already reserved (or the requested seats fall outside the seat plan), the entire reservation must be rejected with an appropriate message. The user must enter `Q` (for the row) to quit the program.

Lesson 27

String manipulation

Purpose of this lesson

In Lesson 7 we saw how to declare and use string variables. We used a trick for chopping off the first letter of a string, namely to input the first character into a `char` variable. In this lesson we see a more general way of referring to individual characters of a string. We also discuss a number of member functions for manipulating string values.

Activity 27.a

Write a program that inputs a string and determines whether it is a palindrome. A palindrome is a string which is identical, irrespective of whether it is read from front to back, or back to front. For example, 'radar', 'abba' and 'abcdeedcba' are palindromes.

Note that the program should be able to work with strings of any length.

Test yourself

We are going to use the fact that we can refer to the individual characters of a string by means of a subscript. For example, the fourth character in the string called `oneWord` is `oneWord[3]`. In other words, we can treat a string like an array of characters.

You will also need to use the `size` member function of the `string` class.

The subactivities below are intended to cover, amongst other things, the concepts required to be able complete the main activity.

Subactivity 27.a.i

Consider the following program:

```
// Split a string into individual characters
#include <iostream>
#include <string>
using namespace std;

int main( )
{
   string oneWord;

   // Prompt for and input a word
```

```
   cout << "Enter a word: ";
   cin >> oneWord;

   // Display the characters in oneWord on separate lines
   cout << "The letters are: " << endl;
   for (int i = 0; i < oneWord.size( ); i++)
      cout << oneWord[i] << endl;

   return 0;
}
```

If the user enters the string `Incredible` when prompted, what will the output of the program be?

Subactivity solution

```
I
n
c
r
e
d
i
b
l
e
```

This program illustrates how we can refer to any character in a string by means of its position or subscript. Note how we use a `for` loop with `i` as a subscript variable. On every iteration, the `i`th character in the string is displayed on a separate line.

The `size` member function of the `string` class gives the length of (i.e. the number of characters in) a `string` object. We use the dot operator between the name of the `string` object and `size` to determine the size of the particular string.

The `for` loop runs from 0 (the subscript of the first character of a string) to 1 less than the size of the string (the subscript of the last character of the string). Think about it. If the string is 10 characters long, the subscript of the first character is 0 and the subscript of the last character is 9.

Discussion

Why are we talking about *member functions*, *classes* and *objects* all of a sudden? What happened to functions, types and variables? Well we are getting a bit more sophisticated now. Strings are not like the simple types that we have seen up to now, namely `int`, `float`, `bool` and `char`. These simple (sometimes called primitive) types are part of the core C++ language, whereas the `string` class is defined in a separate library. That's why we have to add `#include <string>` at the beginning of a program if we intend to use `string` objects in the program.

Member functions are functions that are attached to a class. If you have an object of a class, you can call one of its member functions by using the dot operator. The format of a member function call is

```
ObjectName.MemberFunctionName(Parameters)
```

Note that the `size` member function of the `string` class does not take any parameters, so the parentheses after the member function name are left empty.

Like normal functions, some member functions are void and others return values. The `size` member function of the `string` class returns an integer, namely the length of the `string` object.

This may all seem strange to you, but in fact we have seen something similar before. Can you remember?

Both `cin` and `cout` are actually objects too. They differ from `string` objects in that we don't have to declare them ourselves: `cin` and `cout` are declared in the `<iostream>` header file and are both objects of the `iostream` class. In Subactivity 7.a.iv of Lesson 7, we used the `get` member function with the `cin` object, namely `cin.get( )`.

Subactivity 27.a.ii

Type in the program of the previous subactivity and adapt it to display the characters of a string in reverse order (on the same line).

Subactivity solution

```
// Display the characters of a string in reverse
#include <iostream>
#include <string>
using namespace std;

int main( )
{
   string oneWord;

   // Prompt for and input a word
   cout << "Enter a word: ";
   cin >> oneWord;

   cout << "The reversed word is: " << endl;
   for (int i = oneWord.size( )-1; i >= 0; i--)
      cout << oneWord[i];
   cout << endl;

   return 0;
}
```

Note the boundary values for loop control variable `i` in the `for` loop. The starting value is `oneWord.size( )-1` to ensure that the last character of `oneWord` is displayed first, and the final value is 0 to ensure that the first character is displayed last.

Subactivity 27.a.iii

Write a function to reverse the characters of a string. Note that it shouldn't display any characters. It should take a string as parameter and return the reversed string as the value of the function.

Subactivity solution

```
string reverse(string s)
{
   string answer = "";

   for (int i = 0; i < s.size( ); i++)
      answer = s[i] + answer;
   return answer;
}
```

Of course, we could also let `i` run from the end of the string to the beginning, and concatenate the individual characters onto the end of `answer`.

Subactivity 27.a.iv

Using the techniques illustrated in the previous subactivities, write a boolean function `isPalindrome` that takes a string as parameter and returns `true` or `false` depending on whether the string is a palindrome or not.

Subactivity solution

```
bool isPalindrome(string theWord)
{
   return theWord == reverse(theWord);
}
```

Now write the program for Activity 27.a, incorporating these two functions.

Activity solution

```
// Test whether a string is a palindrome
#include <iostream>
#include <string>
using namespace std;

string reverse(string s)
{
   string answer = "";
   for (int i = 0; i < s.size( ); i++)
      answer = s[i] + answer;
   return answer;
}

bool isPalindrome(string theWord)
{
   return theWord == reverse(theWord);
}

 int main( )
```

```
{
   string oneWord;

   // Prompt for and input a string
   cout << "Enter a single word: ";
   cin >> oneWord;

   // Call isPalindrome and display an appropriate message
   if (isPalindrome(oneWord))
      cout << "It is a palindrome" << endl;
   else
      cout << "No, it isn't a palindrome" << endl;

   return 0;
}
```

Activity 27.b

Write a program to rearrange the title of a book. If the title starts with the word `The`, it should be put at the end of the title, after a comma. For example, if the input is

```
The Great Gatsby
```

the output should be

```
Great Gatsby, The
```

But if the input is

```
Theoretical Computer Science
```

the output should be

```
Title unchanged
```

> **Test yourself**
>
> You need a way of extracting parts of a string to be able to solve this problem. The `substr` member function of the `string` class does this. The two subactivities below show how to use the `substr` member function.

Subactivity 27.b.i

Consider the following program:

```
//Date of birth
#include <iostream>
#include <string>
using namespace std;
```

```
int main( )
{
   string idNum;
   string year, month, day, dateOfBirth;

   cout << "Enter an ID number: ";
   cin >> idNum;
   year = idNum.substr(0,2);
   month = idNum.substr(2,2);
   day = idNum.substr(4,2);
   if (year < "10")
      year = "20" + year;
   else
      year = "19" + year;

   dateOfBirth = day + '/' + month + '/' + year;
   cout << "Date of birth: " << dateOfBirth << endl;

   return 0;
}
```

What will the output be if the input is

```
7605040302001
```

Subactivity solution

The output will be

```
Date of birth: 04/05/1976
```

Discussion

The `substr` member function returns a portion (a substring) of a given string.

There are actually two versions of the `substr` member function: one that takes two parameters and one that only takes one. (Only the two parameter version is used in the above program.)

For the two-parameter version: The first parameter is an integer that indicates the starting position of the substring, and the second parameter, also an integer, indicates how many characters are required. For example, if `idNum` has the value `"7605040302001"`, then `idNum.substr(0,6)` will give the following: From position 0 (remember that the subscripts of a string start at 0) 6 characters from `idNum`, namely the string `"760504"`.

What would `idNum.substr(4,200)` give? In fact it would give the string `"040302001"`. In other words, if the second parameter specifies a substring longer than the characters left in the string, all the remaining characters are returned.

The one-parameter version of `substr` is a better way of doing this. For example, if `idNum` has the value `"7605040302001"`, then `idNum.substr(6)` gives all the characters of `idNum` from position 6 (remember that the subscripts of a string start at 0) to the end, namely the string `"0302001"`.

Note that neither version of the `substr` member function changes the value of the string variable it operates on.

You should now be able to tackle the main activity. You will need to use both versions of the `substr` member function.

Activity solution

```
// Rearrange the title of a book
#include <iostream>
#include <string>
using namespace std;

int main( )
{
   string title, newTitle;

   cout << "Enter the title of a book: ";
   getline(cin, title, '\n');

   if (title.substr(0,4) == "The ")
   {
      newTitle = title.substr(4) + ", The";
      cout << "The new title is: " << newTitle << endl;
   }
   else
      cout << "Title unchanged" << endl;

   return 0;
}
```

Activity 27.c

Write a program that inputs a sentence and two words, and then replaces all occurrences of the first word in the sentence with the second word. An example of the user-dialogue would be:

```
Enter a sentence: The other men there tether the bull
Enter a word to search for: the
Enter a word to replace it with: a
The new sentence is: The oar men are tear a bull
```

Test yourself

This program might seem to be long and complicated, and require a number of functions to be written. In fact, the solution is surprisingly short. We didn't even write any of our own functions to do it.

The `string` class provides a number of powerful member functions (in addition to `size` and `substr`) for manipulating strings, namely `find`, `insert`, `erase` and `replace`. Using these, it is possible to write the entire program quite briefly.

Subactivity 27.c.i deals with the `find` member function. Descriptions of the other member functions can be found in Appendix E.

Subactivity 27.c.i

Consider the following program:

```
// Determines whether a given word appears in a given sentence
#include <iostream>
#include <string>
using namespace std;

int main( )
{
   string sentence, oneWord;
   int position;

   // Prompt for and input a sentence and a word

   cout << "Enter a sentence: ";
   getline(cin, sentence, '\n');
   cout << "Enter a word: ";
   cin >> oneWord;

   if (oneWord.size( ) > sentence.size( ))
      cout << oneWord << " is longer than the sentence!" << endl;
   else
   {
      // Find the first occurrence of the word
      position = sentence.find(oneWord);
      if (position != -1)
         cout << oneWord << " appears in position " << position << endl;
      else
         cout << oneWord << " doesn't appear in " << sentence << endl;
   }
   return 0;
}
```

Type in this program and get it to work. Then investigate what the `find` member function does.

In your study notebook, write down what parameters `find` takes, what values it returns in different situations, and describe what it does in general.

Subactivity solution

The find member function takes a string as parameter and returns an integer. It determines the position of this string (specified as the parameter) in a string object. If the string occurs more than once in the string object, the position of the first occurrence is returned. If the string does not occur in the string object, -1 is returned.

Discussion

Note firstly that we have to use the library function `getline` to input a line of text containing spaces.

Note secondly that `find` does not search for a word in a sentence (as we understand the terms "word" and "sentence"), but rather for a substring in a string.

There is an alternative form of the `find` member function, namely one that takes two parameters: a string and an integer. The additional (integer) parameter allows the search to commence at a position other than position 0. For example, `sentence.find(oneWord, 5)` will return the position of the first occurrence of `oneWord` from position 5 on.

Subactivity 27.c.ii

Write a program to count how many times a given word occurs in a sentence. An example of the user dialogue is

```
Enter a sentence: An orange and an over-ripe banana
Enter a word to count: an
The word occurs 5 time(s) in the sentence
```

Hint: You will need to use the two-parameter version of the `find` member function.

Subactivity solution

```
// Count all occurrences of a word in a sentence
#include <iostream>
#include <string>
using namespace std;

int main( )
{
   string sentence, word;
   int count, position;

   // Input a sentence and a words
   cout << "Enter a sentence: ";
   getline(cin, sentence, '\n');
   cout << "Enter a word to count: ";
   cin >> word;
```

```
   // Count each occurrence of the word
   count = 0;
   position = sentence.find(word);
   while (position != -1)
   {
      count++;
      position = sentence.find(word, position+1);
   }

   cout << "The word occurs " << count << " time(s) in the sentence" << endl;

   return 0;
}
```

We use the `find` member function to find the first occurrence of `word` in `sentence`. If there is at least one occurrence, we enter the loop, increment `count` and use the two parameter version of `find` to find the next occurrence. (We use `position+1` as the second parameter of `find` to ensure that the search does not count the same occurrence twice.) As soon as no occurrence is found, `find` returns -1, which causes the `while` loop to terminate.

Now see if you can complete the main activity. Take a look at Appendix E which contains a few other member functions of the `string` class. You can use `insert` and `erase`, but `replace` is the easiest.

Activity solution

We give two solutions to this problem; one that uses `insert` and `erase`, and another that uses `replace`.

First solution:

```
// Replaces all occurrences of a word in a sentence with another word
#include <iostream>
#include <string>
using namespace std;

int main( )
{
   string sentence, word1, word2;
   int position;

   // Input a sentence and two words
   cout << "Enter a sentence: ";
   getline(cin, sentence, '\n');
   cout << "Enter a word to search for: ";
   cin >> word1;
   cout << "Enter a word to replace it with: ";
   cin >> word2;

   // Search for the first word and replace all occurrences
   // of it with the second word
   position = sentence.find(word1);
```

```
   while (position != -1)
   {
      sentence.erase(position, word1.size( ));
      sentence.insert(position, word2);
      position = sentence.find(word1);
   }

   // Display the new sentence
   cout << "The new sentence is: " << sentence << endl;

   return 0;
}
```

Second solution:

```
// Replaces all occurrences of a word in a sentence with another word
#include <iostream>
#include <string>
using namespace std;

int main( )
{
   string sentence, word1, word2;
   int position;

   // Input a sentence and two words
   cout << "Enter a sentence: ";
   getline(cin, sentence, '\n');
   cout << "Enter a word to search for: ";
   cin >> word1;
   cout << "Enter a word to replace it with: ";
   cin >> word2;

   // Search for the first word and replace all occurrences
   // of it with the second word
   position = sentence.find(word1);
   while (position != -1)
   {
      sentence.replace(position, word1.size( ), word2);
      position = sentence.find(word1);
   }

   // Display the new sentence
   cout << "The new sentence is: " << sentence << endl;

   return 0;
}
```

Note that we don't use the two-parameter version of `find` since we assume that the previous occurrence of `word1` has been replaced.

Important points in this lesson

Programming concepts

We can refer to the individual characters of a string by using a subscript in square brackets after the name of the `string` object. For example, `oneWord[4]` refers to the fifth character in the string `oneWord`. This is because the subscript of a string starts at 0.

The `string` class has a number of member functions (i.e. functions associated with the class) that we can use to manipulate string values. To call a `string` member function, we use the dot operator between the `string` object and the member function name:

```
StringObject.MemberFunctionName(Parameters)
```

Some member functions change the value of the `string` object on which they operate, and others do not. We call member functions that change an object *mutators* and member functions that do not, *accessors*. For example, `size`, `substr` and `find` are accessors, and `insert`, `erase` and `replace` are mutators.

The `size` member function

Format: *StringObject*`.size( )`

Operation: This function returns an integer representing the number of characters in the string *StringObject*.

Example:
```
sentence = "How many?";
length = sentence.size( );
```
will assign the value 9 to `length`.

The `substr` member function

Format: *StringObject*`.substr(`*StartPos*`, `*Length*`)`

Operation: This function starts at position *StartPos* and returns a substring, *Length* characters long, from *StringObject*. *StartPos* and *Length* are integers. If *Length* specifies more characters than there are left from *StartPos* onwards, all the remaining characters are returned.
An alternative version of `substr` only takes a single parameter, namely *StartPos*, and returns all characters in the string from *StartPos* to the end.

Example:
```
sentence = "This is copied";
word = sentence.substr(0, 4);
```
will assign the value `"This"` to `word`.

The `find` member function

Format: *StringObject*`.find(`*Substring*`)`

Operation: This function searches from left to right for the first occurrence of *Substring* in *StringObject*. If found, the function returns the position of the first character of the first occurrence. If not found, the function returns -1.
An alternative version of `find` has two parameters. The optional second parameter is an integer representing the starting position for the search.

Example:
```
sentence = "This is it";
position = sentence.find("is");
```
will assign the value 2 to `position`, and
```
position = sentence.find("is", 6);
```
will assign the value -1 to `position`.

The `insert` member function

Format: *StringObject*`.insert(`*InsertPos*`, `*Substring*`);`

Operation: This void function inserts *Substring* at *InsertPos* in *StringObject*.

Example:
```
sentence = "No sense";
sentence.insert(2, "w it makes");
```
will change the value of `sentence` to `"Now it makes sense"`.

The `erase` member function

Format: *StringObject*`.erase(`*StartPos*`, `*Length*`);`

Operation: This void function starts at position *StartPos* and erases *Length* characters from *StringObject*.

Example:
```
sentence = "Toffee apples";
sentence.erase(2, 9);
```
will change the value of `sentence` to `"Toes"`.

The `replace` member function

Format: *StringObject*`.replace(`*StartPos*`, `*Length*`, `*Substring*`);`

Operation: This void function starts at *StartPos* and replaces *Length* characters of *StringObject* with *Substring*.

Example:
```
sentence = "Data structures";
sentence.replace(1, 9, "en");
```
will change the value of `sentence` to `"Dentures"`.

Programming principles

It's possible to write your own functions to manipulate strings. For example, you could write your own function to determine the position of one string in another string. But why do it? Rather use the member functions that are provided for the `string` class. You can save yourself a lot of time and effort by getting to know the available member functions, to recognise when they can be used.

Exercises

Exercise 27.1

Write a function that converts a string to uppercase. The function should take a string as parameter and return the converted string as the value of the function. All uppercase and non-alphabetic characters should be left unchanged in the string. Hint: Use the `toupper` library function on every character in the string.

Exercise 27.2

Write a program that inputs a person's first names and surname, and then displays just the initials. For example, if the input is `Peter John Smith`, the initials `PJS` must be displayed.

Exercise 27.3

Write a program which inputs a sentence and then displays each word of the sentence on a separate line.

Exercise 27.4

There is a bug in both solutions to Activity 27.c above. What will happen if the user tries to replace a word with the same word, for example replace `the` with `the`? Or what if the second word contains a copy of the first, for example replace `the` with `other`? The program will go into an infinite loop.

Fix this problem.

Lesson 28

Structs

Purpose of this lesson

We have already seen one data structure, namely the array. The second important data structure is a *struct*. (By the way, *struct* is the C++ name for what is generally called a *record* in Computer Science.) With an array, we store a group of data elements of the same type. With a struct, we store a group of data elements of different types.

We often need to work with data values that belong together, but that are of different data types. Consider for instance your own personal information such as your name (string), your age (integer), your height (floating point number), etc. They are values of different types, so we can't store them in an array. Such related data items representing different things can be organised and stored together in a struct. In this lesson we are going to look at the definition of a struct, how values are stored in a struct and how structs are used in a program.

Activity 28.a

Write a program that processes the following information about a movie:

> The title, the director, the year made and a list of the main actors. (If there are more than five main actors, include only the five most important ones.)

The program must input a movie's information and display the information on the screen. It must also determine if the director was also one of the main actors and display an appropriate message.

Your program must use a struct to store the information for the movie, and must use functions as appropriate.

Test yourself

If you know how to handle structs in a program and you could write a program to solve this problem, look at our solution and compare it with yours.

In our solution we used void functions to input the information into a struct and output the information in the struct. Then we used a value-returning function (`bool`) to determine whether the director of the film stored in the struct is also one of the actors.

Although you could write the program without using functions, it is important to know how to pass structs as parameters, so we recommend that you write your program this way as well.

If your program is not correct or you have never worked with structs before, work through the subactivities below.

Subactivity 28.a.i

Consider the following program and answer the questions below it:

```
1  //Example of a struct
2  #include <iostream>
3  using namespace std;
4
5  struct Shirt
6  {
7     int size;
8     char style;
9     float price;
10 };
11
12 int main( )
13 {
14    Shirt shirt1;
15    cout << "Enter the size of the shirt: ";
16    cin >> shirt1.size;
17    cout << "Enter the style of the shirt (A/B/C): ";
18    cin >> shirt1.style;
19    cout << "Enter the price of the shirt: R";
20    cin >> shirt1.price;
21    return 0;
22 }
```

(i) What is the purpose of lines 5 to 10?
(ii) What does line 14 do?
(iii) What is interesting about lines 16, 18 and 20?

(Before looking at the solution, think of the answers. Take a guess, if necessary.)

Subactivity solution

(i) Lines 5 to 10 are a definition of the `Shirt` struct. A `Shirt` struct stores three pieces of information: A `size` (an integer), a `style` (a character) and a `price` (a floating point value).
(ii) Line 14 declares a variable called `shirt1` of type `Shirt`.
(iii) Lines 16, 18 and 20 input and store the `size`, `style` and `price` of a shirt in the `shirt1` struct.

Discussion

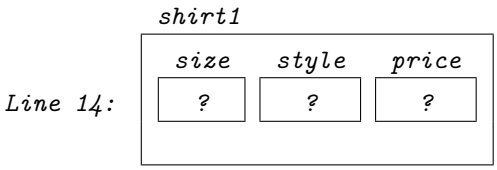
A struct definition (like the one above) does not declare a variable or set aside any memory. It just specifies what memory will be needed when a variable of that type is declared.

Note that we use an uppercase letter for the first letter of the name of the struct type, but a lower case letter for the name of a struct variable. Note the semicolon after the struct definition. This is essential. If you leave out the semicolon, the compiler will complain.

`size`, `style` and `price` are called *fields* of the `Shirt` struct.

Only when a struct variable is declared is memory allocated; enough for all the fields. In this case, memory is put aside for an integer, a single character and a floating point value.

For example, we can illustrate the effect of line 14 with the following variable diagram:



We could have declared more than one variable of type `Shirt` in the program, in which case memory would be put aside for all fields for each `Shirt` variable.

Note the interesting way in which we refer to the individual fields of a struct: We use the name of the struct, followed by the dot operator (a full stop), followed by the name of the field we want to refer to. You can read `shirt1.style` as "the style field of shirt1" or simply "shirt1-dot-style".

If we want to assign the style `'A'` to a shirt, we can write:

```
shirt1.style = 'A';
```

Or, as in the program above, we can input and store a value in a field of a struct by means of a `cin` statement, for example:

```
cin >> shirt1.size;
```

Similarly, we can output the value of a field with a `cout` statement:

```
cout << shirt1.price;
```

In fact, we can draw a series of variable diagrams for struct variables just as we have done for simple data types.

In general, the values of the fields of a struct are input one by one. It is possible to input data into a struct as a whole, as in the statement:

```
cin >> shirt1;
```

but this involves overloading the stream insertion operator. This is not in the scope of this guide.

We can assign one complete struct to another struct of the same type with an assignment statement. Suppose we have two variables `shirt1` and `shirt2` of type `Shirt`. If we have already assigned values to the fields of `shirt1`, we can use the following statement to copy the values:

```
shirt2 = shirt1;
```

Subactivity 28.a.ii

Consider the following function for displaying the data of a struct, and answer the questions below it:

```
void displayData(const Shirt & shirtP)
{
   cout << "Shirt info" << endl;
   cout << "==========" << endl;
   cout << "Style: " << shirtP.style << endl;
   cout << "Size: " << shirtP.size << endl;
   cout << "Price: R " << shirtP.price << endl;
}
```

How does passing a struct as parameter compare with passing a primitive type (e.g. `int`, `char`, `float`) and passing an array as parameter?

In many situations, you can get by without defining a struct in a program by simply declaring separate variables for each of the fields. What advantage of declaring a struct is illustrated by this function?

Give a calling statement for the above function.

Subactivity solution

Passing a struct is like passing a primitive type - not like passing an array. A struct can be passed by reference (using the `&` operator) to allow the function to change its value. Remember that all arrays are passed by reference and we don't need the `&` operator.

If we were to declare separate variables for each of the fields, we'd have to pass all the variables as parameters to the function. By bundling them all together into a struct, we only need to pass a single struct as parameter.

A calling statement for the function would be

```
displayData(shirt1);
```

Subactivity 28.a.iii

Change the struct definition in the program above to also store the colour of a shirt (as a string).

Move the statements to input the data from the `main` function to a separate function called `inputData` and call it in the `main` function. Add the necessary statements to this code to input the colour data as well.

Insert the `displayData` function in the program, and call it in the `main` function. Also add the necessary statements to this function to display the colour data as well.

Subactivity solution

```
//Example of a struct
#include <iostream>
#include <string>
using namespace std;
```

```
struct Shirt
{
   int size;
   char style;
   string colour;
   float price;
};

void inputData(Shirt & shirtP)
{
   cout << "Enter the size of the shirt: ";
   cin >> shirtP.size;
   cout << "Enter the style of the shirt (A/B/C): ";
   cin >> shirtP.style;
   cout << "Enter the colour of the shirt: ";
   cin >> shirtP.colour;
   cout << "Enter the price of the shirt: R";
   cin >> shirtP.price;
}

void displayData(const Shirt & shirtP)
{
   cout << "Shirt info" << endl;
   cout << "==========" << endl;
   cout << "Style: " << shirtP.style << endl;
   cout << "Size: " << shirtP.size << endl;
   cout << "Colour: " << shirtP.colour;
   cout << "Price: R " << shirtP.price << endl;
}

int main( )
{
   Shirt shirt1;

   inputData(shirt1);
   displayData(shirt1);

   return 0;
}
```

Discussion

Yes, you guessed it! We use a plain reference parameter to allow `inputData` to change the data members of the `shirtP` struct (which is a temporary name for the `shirt1` struct). If you run the program, you will see that the data entered for `shirtP` in `inputData` is output for `shirtP` in `displayData`.

Subactivity 28.a.iv

Write the program for the main activity now. Here are some hints:

The definition for the struct containing the information about a movie should have fields for the title, the director, the year made and a list of the 5 main actors. (Hint: A field of a struct may be an array.)

Write three additional functions (apart from the `main` function):

- `inputData` (a void function) should input and store information in a `Movie` struct. After inputting each actor, it should ask the user if there are any more actors.
- `displayData` (a void function) should output the information stored in a `Movie` struct
- `directorActor` (a value-returning function) should determine whether one of the actors and the director of a movie is the same person, and should return `true` or `false`.

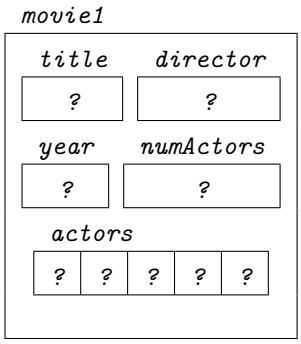
Activity solution

```
// Process a movie
#include <iostream>
#include <string>
using namespace std;

const int MAX_ACTORS = 5;
struct Movie
{
   string title;
   string director;
   int year;
   string actors[MAX_ACTORS];
   int numActors;
};

void inputData(Movie & movieP)
{
   int i;
   char answer;

   cout << "Enter the movie's information:" << endl;
   cout << "Title: ";
   getline(cin, movieP.title, '\n');
   cout << "Director: ";
   getline(cin, movieP.director, '\n');
   cout << "Year: ";
    cin >> movieP.year;
   cout << "The Actors:" << endl;
   i = 0;
   do
   {
      cout << "Actor no " << i+1 << ": ";
      cin.get( );
```

```
      getline(cin, movieP.actors[i], '\n');
      i++;
      if (i < MAX_ACTORS)
      {
         cout << "More actors? (Y/N) ";
         cin >> answer;
      }
   } while ((answer == 'Y' || answer == 'y') && i < MAX_ACTORS);

   movieP.numActors = i;
}

void displayData(const Movie & movieP)
{
   cout << "Movie info" << endl;
   cout << "==========" << endl;
   cout << "Title: " << movieP.title << endl;
   cout << "Director: " << movieP.director << endl;
   cout << "Year: " << movieP.year << endl;
   cout << "Actors:" << endl;
   for (int i = 0; i < movieP.numActors; i++)
      cout << '\t' << movieP.actors[i] << endl;
}

bool directorActor(const Movie & movieP)
{
   for (int i = 0; i < movieP.numActors; i++)
      if (movieP.actors[i] == movieP.director)
         return true;
   return false;
}

int main( )
{
   Movie movie1;
   inputData(movie1);
   displayData(movie1);
   if (directorActor(movie1))
      cout << "The director was also an actor." << endl;
   else
      cout << "The director wasn't an actor." << endl;
   return 0;
}
```

Discussion

Note in the definition of the `Movie` struct that the `actors` field is an array containing 5 elements. The following variable diagram represents the state of the `movie1` instance of the `Movie` struct, immediately after it is declared (in the first statement of the `main` function):



To refer to the array within the struct, we use `movie1.actors`. To refer to the individual elements of this array, we use subscripts like any other array: `movie1.actors[0]` gives us the first element of the array and `movie1.actors[4]` the last element. The field `numActors` is used to store the actual number of places used in array `actors`.

The `inputData` function uses a `do..while` loop because there should be 1 or more actors, i.e. the loop should be executed at least once. The loop is repeated while there are more actors and if less than 5 actors have already been entered.

The rest of the program is fairly straightforward.

Important points in this lesson

Programming concepts

The general layout of a struct definition is as follows:

```
struct StructName
{
   Type1 FieldName1;
   Type2 FieldName2;
   :
   TypeN FieldNameN;
};
```

Some of the fields may have the same data types, so it is more accurate to describe the layout as follows:

```
struct StructName
{
   Type1 FieldList1;
   Type2 FieldList2;
   :
   TypeM FieldListM;
};
```

where every *FieldList* contains the names of one or more fields of the same type, separated by commas.

A struct definition does not allocate any memory. This has to be done with a declaration statement, for example:

```
StructName VariableName;
```

We refer to a specific field of a struct with *VariableName.FieldName*.

We input and output the fields of a struct individually (unless you know how to overload the stream insertion and extraction operators).

We can assign all the values of one struct (say `struct1`) to another struct of the same type (say `struct2`), with a single assignment statement such as:

```
struct2 = struct1;
```

This saves having to copy the fields one by one.

The type of a field may be any valid C++ data type - even an array or another struct.

One characteristic of structs that was not mentioned in the lesson is that the same field name may be used in the definitions of two different struct types in a program. For example

```
struct Shirt
{
   int size;
   char style;
   float price;
};
```

```
struct Trouser
{
   int legSize, waistSize;
   char style;
   float price;
};
```

The field name is then relative to the type of the variable being used, i.e. `shirt1.price` is different from `trouser1.price`.

Programming principles

Use a struct if you have related data (probably not of the same type), especially if you would otherwise have to pass a lot of separate data values to a function.

Our convention is to use an uppercase letter for the first letter of the name of a struct type, but a lowercase letter for the first letters of the names of the fields and for the names of any struct variables.

Exercises

Exercise 28.1

Suppose we have the following struct definition:

```
struct Time
{
   int hours, minutes, seconds;
};
```

Write a void function `changeTime` that receives two parameters: `timeP` of type `Time` and `interval` of type `int` (where `interval` is a number of seconds). The function must change the values of the fields in `timeP` by adding `interval` to them.

In other words, if the value of `timeP` is 1 hour, 30 minutes and 10 seconds and the value of `interval` is 80 seconds, the function must change the values of the fields of `timeP` to 1 hour, 31 minutes and 30 seconds, respectively.

Test your function with a simple program.

Exercise 28.2

ABC Airlines have had a lot of trouble lately with luggage going astray, particularly bags being stolen from carousels at the arrivals halls. They have decided to offer a special service to their passengers to deliver their luggage to their final destination address (up to 100km from the airport). Apart from avoiding the possibility of luggage being stolen from carousels, this will save passengers from having to transport their luggage from the airport to their final destination.

When passengers check-in, they will be presented with a computer console where they will be required to choose whether they will pick up their luggage at the airport (carousel) or would like to have it delivered to their destination. If they choose to have it delivered, they will be required to enter their final destination address. An example of the user dialogue is:

```
ABC Airlines
============
Passenger: Josephine Soap
Flight: ABC123
Destination: JNB - OR Tambo International
Mass of luggage: 20 kg
============
Is all the above information correct (Y/N)? Y
============
We provide a FREE delivery service - up to 100 km from airport.
Would you like your luggage delivered (Y/N)? Y
Enter the address of your final destination (four lines):
15 Albatross Ave
Wonderboom South
Pretoria
0084
Thank you!
Your luggage will be delivered within 3 hours of arrival.
============
```

The program will then print a tag to be attached to the luggage with the passenger name, flight number, destination, mass and address. If the passenger chooses not to have the luggage delivered, the message

```
*** PICK UP AT AIRPORT ***
```

will be printed instead of the address.

Note: The passenger does not enter the name, flight number, destination or mass of the luggage. These are obtained from the ticketing and weighing system used at check-in.

If this information is incorrect (i.e. the user enters `N` when asked), the program will request the passenger to notify the check-in staff and terminate.

Write a program to perform the above user interaction. Your program must call the following dummy function to obtain the passenger's name, flight number and destination, and mass of the luggage:

```
void getPassengerInfo(TagInfo & t)
{
   t.name = "Josephine Soap";
   t.flight = "ABC123";
   t.destination = "JNB - OR Tambo International";
   t.mass = 20.0;
}
```

The body of this function will be replaced later with code that obtains the necessary information from the ticketing and weighing system.

Note that the function stores data in a struct type called `TagInfo`. You must define this struct. It should have fields for the four data values used in the `getPassengerInfo` function, as well as a field to store whether the passenger wants to have luggage delivered, and four fields for the delivery address.

These additional fields of the struct must be updated by another function that you should write to perform the user interaction.

Finally the program should print out the tag. Write a function to display (on the screen) all the information in the struct in a neat format to fit on a tag.

Hint: Start by writing down a list of steps that need to be performed by your program. Then decide what functions you will need to perform the steps. (You might find it helpful to define some additional functions from the ones explained above.)

Lesson 29

Arrays of structs

Purpose of this lesson

In the previous lesson we looked at structs. We saw that a field of a struct may be an array. This lesson looks at the use of arrays to store multiple structs.

Activity 29.a

A certain class has 10 students. Each student has a name and student number, and marks that were awarded for 4 assignments.

Write a program that inputs the names, student numbers and 4 marks for each of the 10 students, calculates the average mark for each student and then the average for the class as a whole. Display the information of all the students with an above average mark.

Test yourself

One way to tackle this problem would be to declare a two-dimensional array, with 10 rows and 4 columns to store all the marks, and then declare a separate array of strings to store the names and/or student numbers of each of the 10 students.

This would not be the best solution however, because the marks really belong with the name and student number of each student. (It is always a good principle to "keep together what belongs together".) In other words, it would be better to define a struct for the information of a student, and then declare an array of 10 structs for the class.

There is nothing really new that you need to know to be able to write this program, so we encourage you to try it on your own. If you can write the program, compare your program with our solution at the end of the lesson and make sure that yours is correct. If you struggled to write the program, work through the following subactivities to arrive at the solution step by step.

Subactivity 29.a.i

Write down the definition of a struct containing the information for a single student: Name, student number, and a list of marks for four assignments.

Subactivity solution

```
const int NUM_MARKS = 4;
struct Student
{
   string name, studNo;
   int marks[NUM_MARKS];
};
```

Subactivity 29.a.ii

Using the definition above, write the following functions:

- a void function that inputs data into such a struct passed as a parameter, and
- a void function that receives such a struct as parameter and displays the information,
- a function that receives such a struct as parameter and returns the student's average.

Type these into a program skeleton.

Subactivity solution

The following functions should be typed into a program skeleton together with the struct definition.

```
void inputData(Student & studentP)
{
   cout << "Enter the student's information" << endl;
   cout << "Student number: ";
   cin >> studentP.studNo;
   cout << "Name: ";
   cin >> studentP.name;
   cout << "Marks:" << endl;
   for (int i = 0; i < NUM_MARKS; i++)
   {
      cout << "Assignment " << i << ": ";
      cin >> studentP.marks[i];
   }
}

void displayData(const Student & studentP)
{
   cout << "The student's information:" << endl;
   cout << "Student number: " << studentP.studNo << endl;
   cout << "Name: " << studentP.name << endl;
   cout << "Marks:" << endl;
   for (int i = 0; i < NUM_MARKS; i++)
      cout << "   Assignment " << i << ": " << studentP.marks[i] << endl;
}

float average(const Student & studentP)
```

```
{
   int total = 0;
   for (int i = 0; i < NUM_MARKS; i++)
      total += studentP.marks[i];
   return float(total) / NUM_MARKS;
}
```

Subactivity 29.a.iii

Study the problem statement for the main activity and make a list of all the different tasks the program must complete.

Subactivity solution

1. *For each student, we must input a student number, a name and 4 marks into a struct. We can use the function inputData to do this. There are 10 students, so we will have to use a loop with 10 iterations and call the function from within this loop to input each student's information. Since we need all the students' information later, we store it in an array with 10 elements (structs) of type Student.*

2. *We have to calculate an average mark for each of the students. We therefore need a loop to go through the array. On every iteration we call the average function to calculate the average for a student, and store this in a separate array with 10 elements - one for each student. Instead of doing this in a separate loop, we can do these calculations in the loop described in 1 above (i.e. the loop used to input the information for 10 students).*

3. *Next we must calculate the overall average for all the students. We use the array with the 10 averages to do this (created in the previous step). To calculate the average, we have to add up the individual averages of each student and divide the total by 10. This again requires a loop, but again we can use the same loop that we used to input the students' information. Therefore, as we input the data, we calculate each student's average and add it to a total that will eventually be divided by 10.*

4. *Lastly, we compare each student's average with the class average. If the student's average is greater, we display the student's information (from the student array). This step requires a separate loop. It can't be done in the same loop, because we have to have calculated the overall average, before we can decide whether an individual student's average is above the overall average or not.*

Subactivity 29.a.iv

Write declarations for the following arrays:

- An array with 10 structs as elements.
- An array containing 10 student averages.

Subactivity solution

The declarations should look as follows:

```
const int NUM_STUDS = 10;
Student students[NUM_STUDS];
float averages[NUM_STUDS];
```

Discussion

Each element of the array, `students[0]`, `students[1]`, ..., `students[9]`, is a struct consisting of three fields. For example `students[0]` has the fields `studNo`, `name` and `marks`. To refer to these fields we use the notation `students[0].studNo`, `students[0].name` and `students[0].marks`, respectively.

But the `marks` field is also an array. To refer to individual elements of this field, we use `students[0].marks[0]`, `students[0].marks[1]`, etc. Each of these elements is an integer.

So to refer to the third result of the eighth student, we use `students[7].marks[2]`.

Subactivity 29.a.v

Write down a `for` loop that uses function `inputData` to input the data for 10 students into the array students.

Subactivity solution

```
for (int i = 0; i < NUM_STUDS; i++)
{
   cout << "Student " << i << endl;
   inputData(students[i]);
}
```

Remember that `students[i]` is a struct of type `Student`, and so it is a valid actual parameter in a call to the `inputData` function. Function `inputData` changes the struct parameter passed to it (namely the `i`th element of the array `students`) because it uses a reference parameter.

Subactivity 29.a.vi

Using functions `inputData`, `displayData`, and `average` above, write the `main` function required by the main activity.

Use the solution to Subactivity 29.a.iii as a guideline.

Activity solution

```
//Process student assignment marks
#include <iostream>
#include <string>
using namespace std;

const int NUM_MARKS = 4;
struct Student
{
```

```
   string name, studNo;
   int marks[NUM_MARKS];
};

void inputData(Student & studentP)
{
   cout << "Enter the student's information" << endl;
   cout << "Student number: ";
   cin >> studentP.studNo;
   cout << "Name: ";
   cin >> studentP.name;
   cout << "Marks:" << endl;
   for (int i = 0; i < NUM_MARKS; i++)
   {
      cout << "Assignment " << i << ": ";
      cin >> studentP.marks[i];
   }
}

void displayData(const Student & studentP)
{
   cout << "The student's information:" << endl;
   cout << "Student number: " << studentP.studNo << endl;
   cout << "Name: " << studentP.name << endl;
   cout << "Marks:" << endl;
   for (int i = 0; i < NUM_MARKS; i++)
      cout << "   Assignment " << i << ": " << studentP.marks[i] << endl;
}

float average(const Student & studentP)
{
   int total = 0;
   for (int i = 0; i < NUM_MARKS; i++)
      total += studentP.marks[i];
   return float(total) / NUM_MARKS;
}

int main( )
{
   const int NUM_STUDS = 10;
   Student students[NUM_STUDS];
   float averages[NUM_STUDS];
   float grandTotal = 0;
   float classAverage;

   //Input data and calculate average for each student
   for (int i = 0; i < NUM_STUDS; i++)
   {
      cout << "Student " << i << endl;
      inputData(students[i]);
      averages[i] = average(students[i]);
      grandTotal += averages[i];
   }
```

```
   //Calculate and display class average
   classAverage = grandTotal / NUM_STUDS;
   cout << "The class average is " << classAverage << endl;

   //Display all students who did above average
   cout << endl << "These students did above average:" << endl;
   for (int i = 0; i < NUM_STUDS; i++)
      if (averages[i] > classAverage)
         displayData(students[i]);

   return 0;
}
```

Discussion

In each repetition of the first `for` loop, we do the following:

- input data for a student and store it in `students[i]`,
- send this struct to function `average` to calculate the student's average and store the returned result in array `averages`,
- add the student's average to the running total of all the averages.

In the next step we use the total that was calculated in the `for` loop and calculate the average for the class as a whole.

In the last `for` loop we compare the `i`th student's average with the class average. If the student's average is better, we display his/her information (stored in `students[i]`) using the `displayData` function.

Note that we don't need a declaration of a single student struct in the `main` function.

Important points in this lesson

Programming concepts

In this lesson we looked at the use of arrays of which the elements are structs. An array of structs is declared as follows:

```
StructName ArrayName[NUM_ELEMS];
```

Each element of array *ArrayName* (i.e., *ArrayName*[0], *ArrayName*[1], etc.) is a struct of type *StructName*. To access the fields of the individual structs, we use the dot operator, e.g. *ArrayName*[*Index*].*FieldName1*, where *Index* is a subscript between 0 and *NUM_ELEMS* - 1.

Programming principles

When you find you are having to define parallel arrays (i.e. arrays with the same number of elements) think whether you can't use a struct to store the data of corresponding elements together.

Exercises

Exercise 29.1

Write a program that provides an analysis of the production and sales data of items handled by a company called *LekkerKoop Ltd*. The following information must be stored for each kind of item: description, number produced, number sold, and the difference between the number produced and the number sold. The data for all the items must be stored in an array. The maximum number of kinds of items is 100. The program should output a clear report indicating the items for which production exceeded sales. The program may input the number of kinds of items that must be processed from the keyboard.

Exercise 29.2

Adapt the solution to the main activity of this lesson to avoid using parallel arrays. (Arrays are called *parallel arrays* when they store the same number of elements and corresponding elements store related data.) You'll need to do the following:

- Add a field called `average` to the `Student` struct to store the average.
- Change the `average` function to calculate and store the average in the `average` field. (It should therefore be changed to a void function, and its parameter should be passed by reference.)
- Make the necessary changes to the `main` function to access the new field.

Lesson 30

Classes

Purpose of this lesson

The purpose of this study guide has been to provide the foundations necessary to learn object-oriented programming. Up to now, the only object-oriented programming that we have encountered has been in the `string` class which has member functions.

Although it is not the purpose of this study guide to explain object-oriented programming, this final lesson shows how the concepts that we have covered form the building blocks of object-oriented programming. In particular, we show how the class structure of C++ is an extension of a struct. Simply stated, a class consists of data combined with a number of functions (that we call member functions) that act on that data.

Activity 30.a

Write a program that inputs and processes transactions on a banking account. The types of transactions are deposit, balance enquiry and withdrawal. Every time a transaction is performed, a fee should be charged (and subtracted from the balance) as follows: R1.00 per deposit, R0.50 per balance enquiry and R1.50 per withdrawal. However, if the balance is negative after a withdrawal (i.e. the account is overdrawn) the withdrawal fee should be R5.00. At the end of the list of transactions (for a month, say), the program should display all the transactions, the total of all the fees charged and the closing balance of the account.

The program should use the following codes for transactions: D for deposit, B for balance enquiry and W for withdrawal. You can assume that there aren't more than 30 transactions per month.

An example of the user interaction would be

```
Enter the transactions for the month
(D)eposit, (B)alance enquiry, (W)ithdrawal, E(X)it:
D 100
W 50
B
W 50
D 100
X

Monthly statement
=================
Deposit  R100.00
Withdrawal R50.00
Balance enquiry
Withdrawal R50.00
Deposit R100.00
```

```
Total fees R8.00
-----------------------
Closing balance R91.00
```

NB: In your program, the data should be stored in such a way that neither the balance nor any of the details of any of the transactions can be inadvertently changed. The only way the balance or the transactions should be able to be changed should be by calling one of three functions, namely `deposit`, `balanceEnquiry` and `withdrawal`.

Test yourself

You should be able to write the above program using structs to store the data for an account, and the data for the transactions. The problem comes with the requirement that the data should only be changed by the three specified functions. The structs that we defined in the previous lessons allowed any part of the program to access and change any of its fields.

We can protect data from inadvertent (or malicious) damage by defining a class, and specifying which functions can access the data held in the class. The fields of a class are called *member variables*. The functions that operate on the member variables are called *member functions*.

The first two subactivities below show how a class is defined in C++. The remaining subactivities show how to define the classes necessary to complete the main activity.

Subactivity 30.a.i

Study the following program (which is an alternative solution to the main activity of Lesson 28 but uses a class instead of a struct) and answer the questions below it.

```
// Process a movie
#include <iostream>
#include <string>
using namespace std;

const int MAX_ACTORS = 5;
class Movie
{
  public:
    void inputData( );
    void displayData( );
    bool directorActor( );
  private:
    string title;
    string director;
    int year;
    string actors[MAX_ACTORS];
    int numActors;
};
```

```
void Movie::inputData( )
{
   int i;
   char answer;

   cout << "Enter the movie's information:" << endl;
   cout << "Title: ";
   getline(cin, title, '\n');
   cout << "Director: ";
   getline(cin, director, '\n');
   cout << "Year: ";
   cin >> year;
   cout << "The Actors:" << endl;
   i = 0;
   do
   {
      cout << "Actor no " << i+1 << ": ";

      cin.get( );
      getline(cin, actors[i], '\n');
      i++;
      if (i < MAX_ACTORS)
      {
         cout << "More actors? (Y/N) ";
         cin >> answer;
      }
   } while ((answer == 'Y' || answer == 'y') && i < MAX_ACTORS);

   numActors = i;
}

void Movie::displayData( )
{
   cout << "Movie info" << endl;
   cout << "==========" << endl;
   cout << "Title: " << title << endl;
   cout << "Director: " << director << endl;
   cout << "Year: " << year << endl;
   cout << "Actors:" << endl;
   for (int i = 0; i < numActors; i++)
      cout << '\t' << actors[i] << endl;
}

bool Movie::directorActor( )
{
   for (int i = 0; i < numActors; i++)
      if (actors[i] == director)
         return true;
   return false;
}

int main( )
{
```

```
   Movie movie1;
   movie1.inputData( );
   movie1.displayData( );
   if (movie1.directorActor( ))
      cout << "The director was also an actor." << endl;
   else
      cout << "The director wasn't an actor." << endl;
   return 0;
}
```

1. How does the class definition differ from the struct definition as given in the solution to Activity 28.a?
2. What do your think the terms `public` and `private` in a class definition mean? (Take a guess.)
3. How do the member functions of the `Movie` class differ from the functions that we wrote in the solution to Activity 28.a?
4. How does the declaration of an object differ from the declaration of a struct variable? (See `main` function.)
5. How do the member function calls of an object differ from the function calls in the `main` function of the solution to Activity 28.a?

Subactivity solution

1. The class definition starts with the reserved word `class` rather than `struct`. A number of member function headers are included in the class definition. There are two reserved words `public` and `private` used before the list of member functions and the list of member variables, respectively.
2. Public means that something is available for everyone to use, whereas private means that it is hidden away. Since the member functions are all listed under `public`, this indicates that they are available to the rest of the program, whereas the member variables are inaccessible.
3. There are three differences between these member functions and the functions used previously: (i) The member functions don't have any parameters. (ii) In the definition of each member function, `Movie::` is used before the name of the member function. (iii) In the body of each member function, the names of member variables are not preceded by the name of a `Movie` object and the dot operator. They are used on their own.
4. It looks exactly the same!
5. Instead of passing a `Movie` struct as parameter to a function, we use a `Movie` object, followed by the dot operator, followed by the member function name.

Discussion

The answers to the five questions above are all very important. We discuss each one:

1. As stated in the **Purpose of this lesson**, a class combines data and functions to operate on that data. You can see this clearly in the definition of the `Movie` class. This is what we mean by encapsulation.

2. Encapsulation not only refers to this combination of data and functions, but also to the fact that the data is hidden, i.e. inaccessible, to the rest of the program. The only access to the data is through the member functions. That's why we use the access specifiers `public` and `private`.
3. (i) The reason why the member functions don't need to take a `Movie` object as parameter is that the member functions are connected to the `Movie` class. Member functions of a class are always called with a specific object of that class (see discussion of point 5) and so the member function operates on that specific object. (ii) The `Movie::` prefix indicates that a function is a member function of the `Movie` class. The double colon is called the *scope resolution operator*. (iii) As stated above, a member function is called on a specific object. When a member function refers to a member variable, it refers to the member variable of that specific object. (Once again, see point 5. below.)
4. The declaration of an object looks exactly the same as the declaration of a struct variable. Like a struct, memory is only allocated when an object is declared. The definition of a class does not reserve any memory.
5. A member function call is always attached to an object of the relevant class. The member function operates on that specific object. Any references to the member variables of the class in the body of the member function are interpreted as references to member variables of that specific object.

Subactivity 30.a.ii

In Lesson 27 we discussed the idea of *accessor* and *mutator* member functions of a class. An accessor is a member function that simply accesses the member variables of an object, but doesn't change them. A mutator changes the member variables of an object. Which of the member functions of the `Movie` class are accessor and which are mutators?

Subactivity solution

`inputData` is a mutator (because it inputs values and stores them in the member variables of the object) but `displayData` and `directorActor` are accessors (because they don't change the member variables of the object).

Discussion

There is a way to indicate and force a member function to be an accessor: we add the reserved word `const` to the end of a member function header. For example, to specify that `displayData` is an accessor we change the class definition to

```
class Movie
{
  public:
    void inputData( );
    void displayData( ) const;
    ... etc.
 };
```

and we change the member function definition to

```
void Movie::displayData( ) const
{
   cout << "Movie info" << endl;
   cout << "==========" << endl;
   ... etc.
}
```

The compiler will now complain if any attempt is made to change the values of any member variables in the body of the accessor.

It is good programming practice to always specify which member functions are accessors by adding `const` in this way.

Subactivity 30.a.iii

Give the definitions of two structs for use by the program for the main activity. (We will be using classes, but this is an intermediate step.) The first struct should store the information for a transaction, and the second struct should store information for an account. Remember a transaction should store a code (for the type of transaction) and an amount. Apart from the balance and the fees charged, an account should keep a record of up to 30 transactions.

Subactivity solution

```
struct Transaction
{
   char type;
   float amount;
};

const int MAX_TRANSACT = 30;
struct Account
{
   float balance;
   Transaction transacts[MAX_TRANSACT];
   int numTransacts;
   float feeTotal;
};
```

Discussion

Note the following about these structs:

- The `Account` struct stores the balance, an array of 30 transactions, the number of actual transactions and the total of all the fees (see explanation below). We have to store the number of

transactions as a field (`numTransacts`) because when we display all the transactions, we need to know how many transactions have been stored in the array. (It might not be 30.)

- We have to define the `Transaction` struct before the `Account` struct because the `Account` struct uses the definition of the `Transaction` struct. If these structs were part of a program, the compiler would complain if the `Transaction` definition was placed after the `Account` definition.
- We store `feeTotal` in the `Account` struct, rather than individual fees for each transaction. We could have done it the other way round (in fact, it might be useful to have a record of each fee charged for each transaction) but since storing individual fees would be less efficient in terms of space and time (think about it), we rather do it this way.

Subactivity 30.a.iv

Now we need to think about converting these struct definitions into class definitions. In particular, we need to decide what member functions are required by each class (including their return types, their parameters and whether they should be accessors or mutators) and specify them in the `public` section of the class definitions.

The problem of the main activity stated that we need three functions `deposit`, `balanceEnquiry` and `withdrawal`. Which struct should they operate on: an `Account` or a `Transaction`?

Subactivity solution

We need an account before we can call `deposit`, `balanceEnquiry` or `withdrawal`, but we don't need a transaction instance. In fact, when we call one of these three functions, it should generate a transaction.

The three functions should therefore be member functions of the `Account` class.

Subactivity 30.a.v

Turn the `Account` struct into a class, and add headers for these three member functions to it. Specify which are accessors by adding the `const` reserved word as appropriate.

Subactivity solution

```
const int MAX_TRANSACT = 30;
class Account
{
  public:
    void deposit(float a);
    float balanceEnquiry( );
    void withdrawal(float a);
  private:
    float balance;
    Transaction transacts[MAX_TRANSACT];
    int numTransacts;
    float feeTotal;
};
```

Discussion

Note firstly that not one of the three member functions are accessors. You might have thought that `balanceEnquiry` should be an accessor, but remember that a fee must be charged for a balance enquiry (so `feeTotal` and `balance` must change) and a balance enquiry is a transaction (so it must be added to `transacts`, the array of transactions).

Note also that `balanceEnquiry` returns a value (namely the current balance of the account). Even though our program won't be using this value (as you can see from the user interaction in the specification of the main activity) we will make `balanceEnquiry` return a value because it is a sensible thing to do, particularly so that the `Account` class can be used in future (say in the program for an ATM machine) where the balance will need to be displayed.

Only `deposit` and `withdrawal` need parameters, in particular a single floating point value specifying the amount being deposited or withdrawn, respectively.

Subactivity 30.a.vi

Aren't there any more member functions that should be added to the `Account` class? What about member functions like `inputData` and `displayData` that we used for the `Movie` class?

Subactivity solution

We can't have a member function like `inputData` because it would definitely change the data in the `Account` class, and the requirements of the problem stated clearly that the only functions that may change the data are `deposit`, `balanceEnquiry` and `withdrawal`. The `main` function (or some other non-member function) will have to do the input and call the appropriate member functions.

However, we will have to implement something like a `displayData` function (say `displayStatement`) since the data is all private, and the only way to get access to it is through some member function.

Subactivity 30.a.vii

Add a header for a `displayStatement` member function to the `Account` class definition.

Subactivity solution

```
const int MAX_TRANSACT = 30;
class Account
{
  public:
    void deposit(float a);
    float balanceEnquiry( );
    void withdrawal(float a);
    void displayStatement( ) const;
  private:
    float balance;
    Transaction transacts[MAX_TRANSACT];
    int numTransacts;
```

```
    float feeTotal;
};
```

Did you remember to specify `displayStatement` as an accessor?

Subactivity 30.a.viii

You are now ready to write implementations for each of the member functions.

Before you proceed, you might be wondering about the `Transaction` struct. Will we be turning it into class, and if so, what member functions will it have?

In fact, we are going to leave it as a struct. This is to keep things simple, but also, as we explain in the discussion at the end of the solution to the main activity, there is no advantage to defining a `Transaction` as a class.

Firstly, write an implementation for the `deposit` member function. (To refresh your memory, you might like to refer back to the member functions defined in Subactivity 30.a.i to see the correct format.)

Subactivity solution

```
void Account::deposit(float a)
{
   balance += a;
   feeTotal += DEPOSIT_FEE;
   balance -= DEPOSIT_FEE;
   transacts[numTransacts].type = 'D';
   transacts[numTransacts].amount = a;
   numTransacts++;
}
```

This function assumes that a global constant called `DEPOSIT_FEE` has been defined somewhere. It is better to put values like this in constants rather than coding the literal values into the program. We could define `DEPOSIT_FEE` as a local constant of the `deposit` member function, but we are going to have to define a few constants in the program. We prefer to define them together in one place so that their values can be changed in one place if necessary.

Remember that `deposit` is a member function of the `Account` class, so it has direct access to all the private data of the `Account` class. When `deposit` is called, it will always be attached to a specific object, say `account1`. The member variables `balance`, `feeTotal`, `transacts` and `numTransacts` will therefore all be relative to (i.e. belong to) that specific `Account` object.

There is something else that should be bothering you about this function, however, namely initialisation. Where will balance, `feeTotal`, `transacts` and `numTransacts` be initialised? What will their values be when `deposit` is called for the first time?

We could consider writing a member function called `initialise` (it would have to be a member function because it needs to change private data of the `Account` class) but we have the requirement that the only member functions that may change the values of member variables are `deposit`, `balanceEnquiry` and `withdrawal`. In any case, it would be dangerous to write an `initialise` member function, because if a programmer called it a second time inadvertently, all the data for an account would be wiped out. So what can we do?

Discussion

The answer is a special member function called a *constructor*. This member function is called implicitly (i.e. without the programmer calling it explicitly) when an object of a class is declared.

The constructor is a strange member function in that is doesn't have a return type at all (neither void not any other type), and its name is always the name of the class. For example, we could define a constructor for the `Account` class as follows:

We first need to add a header for the constructor in the class definition:

```
class Account
{
  public:
    Account( );
    void deposit(float a);
    float balanceEnquiry( );
    void withdrawal(float a);
    ... etc.
};
```

Note the absence of a return type and the name of the constructor. We haven't included any parameters because we don't need any in this case. Note also that the constructor is defined in the `public` section. If we placed it in the `private` section, the compiler wouldn't allow us to declare an `Account` object!

Then we need to implement the constructor below the class definition:

```
Account::Account( )
{
   balance = 0;
   numTransacts = 0;
   feeTotal = 0;
}
```

We could initialise all the values in the transaction array as well, but since we will never assume that the array contains any values, and since `numTransacts` will keep track of how many actual transactions have been stored in the array, this isn't necessary.

But haven't we disobeyed the requirement that the only functions that may change the values of member variables of an `Account` are the three specified ones? Well maybe, but the programmer can't call the constructor again to reset the values of an `Account`. The constructor can only be called once per object, and this is done implicitly during the declaration of an object.

Initialisation of member variables of a class is extremely important. Whenever you define a class you should consider this. The standard way of initialising member variables of an object is to do it in a constructor.

Subactivity 30.a.ix

Now write implementations for the other member functions, namely `balanceEnquiry`, `withdrawal` and `displayStatement`.

Subactivity solution

```
float Account::balanceEnquiry( )
{
   feeTotal += BALANCE_FEE;
   balance -= BALANCE_FEE;
   transacts[numTransacts].type = 'B';
   transacts[numTransacts].amount = 0;
   numTransacts++;
   return balance;
}

void Account::withdrawal(float a)
{
   balance -= a;
   if (balance >= 0)
   {
      feeTotal += WITHDRAWAL_FEE;
      balance -= WITHDRAWAL_FEE;
   }
   else
   {
      feeTotal += OVERDRAWN_FEE;
      balance -= OVERDRAWN_FEE;
   }
   transacts[numTransacts].type = 'W';
   transacts[numTransacts].amount = a;
   numTransacts++;
}

void Account::displayStatement( ) const
{
   cout << endl << "Monthly Statement" << endl;
   cout << "=================" << endl;
   for (int i = 0; i < numTransacts; i++)
   {
      switch (transacts[i].type)
      {
         case 'D':
            cout << "Deposit\tR" << transacts[i].amount << endl;
            break;
         case 'B':
            cout << "Balance enquiry" << endl;
            break;
         case 'W':
            cout << "Withdrawal\tR" << transacts[i].amount << endl;
            break;
         default:
```

```
            cout << "**Invalid transaction code**" << endl;
        }
    }
    cout << "Total Fee\tR" << feeTotal << endl;
    cout << "---------------------------------" << endl;
    cout << "Closing balance\tR" << balance << endl;
}
```

Once again we use constants, namely `BALANCE_FEE`, `WITHDRAWAL_FEE` and `OVERDRAWN_FEE`, instead of literals for the various fees. You should do this whenever possible.

The `balanceEnquiry` member function is very similar to `deposit`. Make sure you understand all the differences.

The `withdrawal` member function is also very similar to `deposit` except that it contains an `if` statement to decide which fee to charge.

The `displayStatement` member function uses a `switch` statement to display the transaction appropriately. (A `default` clause is included in case the data has become corrupted in some way.) The `switch` statement is inside a `for` loop that processes all the actual transactions.

Subactivity 30.a.x

Now, believe it or not, you are ready to write the main function that uses this class and to put all the code together to form the solution to the main activity.

You need to do the following:

- Define all the fee constants.
- Define the `Transaction` struct and the `Account` class.
- Specify the implementations of all the member functions of the `Account` class including the constructor.
- Write the `main` function.

The `main` function should declare an `Account` object and then start inputting the transactions. (You could define a non-member function to input the transactions - we did it in the `main` function.) Use a `switch` statement to determine which of the three member functions to call for each transaction. Finally, call `displayStatement` to display the monthly statement.

Activity solution

```
// Processes a banking account and displays a monthly statement
#include <iostream>
using namespace std;

const float DEPOSIT_FEE = 1.00;
const float BALANCE_FEE = 0.50;
const float WITHDRAWAL_FEE = 1.50;
const float OVERDRAWN_FEE = 5.00;

struct Transaction
```

```
{
   char type;
   float amount;
};

const int MAX_TRANSACT = 30;
class Account
{
  public:
    Account( );
    void deposit(float a);
    float balanceEnquiry( );
    void withdrawal(float a);
    void displayStatement( ) const;
  private:
    float balance;
    Transaction transacts[MAX_TRANSACT];
    int numTransacts;
    float feeTotal;
};

Account::Account( )
{
   balance = 0;
   numTransacts = 0;
   feeTotal = 0;
}

void Account::deposit(float a)
{
   balance += a;
   feeTotal += DEPOSIT_FEE;
   balance -= DEPOSIT_FEE;
   transacts[numTransacts].type = 'D';
   transacts[numTransacts].amount = a;
   numTransacts++;
}

float Account::balanceEnquiry( )
{
   feeTotal += BALANCE_FEE;
   balance -= BALANCE_FEE;
   transacts[numTransacts].type = 'B';
   transacts[numTransacts].amount = 0;
   numTransacts++;
   return balance;
}

void Account::withdrawal(float a)
{
   balance -= a;
   if (balance >= 0)
   {
```

```
      feeTotal += WITHDRAWAL_FEE;
      balance -= WITHDRAWAL_FEE;
   }
   else
   {
      feeTotal += OVERDRAWN_FEE;
      balance -= OVERDRAWN_FEE;
   }
   transacts[numTransacts].type = 'W';
   transacts[numTransacts].amount = a;
   numTransacts++;
}

void Account::displayStatement( ) const
{
   cout << endl << "Monthly Statement" << endl;
   cout << "=================" << endl;
   for (int i = 0; i < numTransacts; i++)
   {
      switch (transacts[i].type)
      {
         case 'D':
            cout << "Deposit\tR" << transacts[i].amount << endl;
            break;
         case 'B':
            cout << "Balance enquiry" << endl;
            break;
         case 'W':

            cout << "Withdrawal\tR" << transacts[i].amount << endl;
            break;
      }
   }
   cout << "Total Fee\tR" << feeTotal << endl;
   cout << "--------------------------------" << endl;
   cout << "Closing balance\tR" << balance << endl;
}

int main( )
{
   Account account1;
   char type;
   float amount;

   cout << "Enter the transactions for the month" << endl;
   cout << "(D)eposit, (B)alance enquiry, (W)ithdrawal, E(X)it:" << endl;
   cin >> type;
   while (toupper(type) != 'X')
   {
      switch(toupper(type))
      {
         case 'D':
            cin >> amount;
```

```
            account1.deposit(amount);
            break;
         case 'B':
            account1.balanceEnquiry( );
            break;
         case 'W':
            cin >> amount;
            account1.withdrawal(amount);
      }
      cin >> type;
   }
   cout << endl;

   account1.displayStatement( );

   return 0;
}
```

Discussion

We have discussed the `Account` class and its member functions, and the `main` function is quite straightforward. All we need to say is something about the `Transaction` struct (as we promised).

The question is how good is our encapsulation? In other words, does the program as it is written now protect an `Account` object, in particular the array of transactions, from being inadvertently or maliciously changed by the program that uses it? We know that a struct does not enforce encapsulation, i.e. the rest of the program can access and change the fields of a struct as it likes. So can't the rest of the program change the details of the transactions as things stand in our solution?

The answer is no. But why, you may ask. The reason is that the array of transactions is private in the definition of the `Account` class. This doesn't make the `Transaction` struct private, it just makes all the transactions in an `Account` private. In other words, if the `main` function declared a variable of type `Transaction`, it could change the values of the fields directly. However, the only way to get at the transactions stored in an `Account` is through array `transacts`, and since `transacts` is private, all the transactions are inaccessible to the rest of the program.

This is quite a complex argument for a situation that doesn't occur very often. As stated earlier, we could have defined `Transaction` as a class, but that would have made the final solution even longer. Since it wasn't necessary (as explained above) we left it as a struct.

In general, when should you define a struct and when should you define a class? Some object-oriented programmers say that you should always use a class. We feel that there are situations that a struct is a better (and simpler) choice. Firstly, if you require a data-only data structure, use a struct. By a data-only data structure, we mean that there aren't operations (i.e. meaningful member functions) that operate exclusively on the data and "belong" to the data. Secondly, if encapsulation (in particular the protection/hiding of data that goes with it) is not necessary, why use a class? Rather use a struct and allow the rest of the program to have direct access to all the data. Finally, if code re-use is not an issue, use a struct. In other words, if you would never need to use the definition of a class and its member functions in another program, there is little advantage in enforcing encapsulation and strict access, since the struct and the program that works with it will always be together.

A final aspect of classes that we have neglected to apply is the use of *separate files*. Generally (and ideally), a class definition and its member functions are placed in separate files from the file containing the program that uses the class. We have placed the definition of a class and its member functions in the same file as the `main` function to keep things simple. There are (at least) two reasons why it is better to put them in separate files. (i) A class can be re-used by another program. If it is incorporated in the same file, you have to copy and paste code to re-use it. (ii) It increases encapsulation. Details that are not necessary for the program that uses a class to know about, are hidden from it.

Important points in this lesson

Programming concepts

A class can be viewed as a glorified struct. The general layout of a class definition is:

```
class ClassName
{
  public:
    MemberFunctionList;
  private:
    MemberVariableList;
};
```

`MemberFunctionList` is a list of member function headers: names of member functions each with return types and parameters as appropriate. Each member function header can be followed by `const` indicating that it is an accessor rather than a mutator. This prevents the member function from changing the values of any of the member variables.

A *constructor* is a special member function of a class that has the name of the class and has no return type. A constructor is called automatically (implicitly) when an object of the class is declared. In general, a constructor is used to initialise the member variables of an object. The `Movie` class that we implemented in this lesson shows that you can define a class without a constructor if you don't need to initialise the member variables. (We provided an `inputData` member function to input and store values in the member variables.)

`MemberVariableList` is a list of member variable declarations: types and names of member variables comprising any valid C++ types.

A class definition is generally followed by definitions of the member functions. These are like common function definitions (with return types and parameters matching those specified in the class definition), except that the member function name is preceded by *`ClassName`*`::`, indicating that the function is a member function of the *`ClassName`* class.

Member functions of a class can access the (private) member variables of the class directly (i.e. a member function can refer to a member variable without prefixing it with an object name and the dot operator). Accessor member functions can only refer to the member variables but not change their values. Mutators can change the values of member variables.

The `main` function (and any other functions that are not member functions of the class) cannot access private member variables of the class at all. The only way they can get hold of or change the values of such member variables is by calling member functions that are provided for the class. This way of protecting the member variables of a class from access and inadvertent or malicious change is called *encapsulation*.

Programming principles

It is important to consider the initialisation of member variables of a class, and in most cases one needs to implement a constructor to do this.

It is good programming practice to indicate which member functions are accessors and which are mutators by using the `const` reserved word at the end of the headers of accessors. This makes it clear to anyone who reads the code that the member function will not change the values of any of the member variables.

When you have a number of functions that work with structs, it is generally better to define a class and make the functions member functions of the class. In this way, data and the functions that work with it are encapsulated together. If the data is made private in the class, the encapsulation protects the data from being changed in incorrect ways by the rest of the program. It can force the program to only access the data through the given member functions.

In general, to decide whether to use a struct or a class, the question is if there are functions that work exclusively on the data, and that "belong" to the data. If so, a class is probably better than a struct.

Exercises

Exercise 30.1

Rewrite the solution to Activity 29.a using a class instead of a struct. Specify which member functions are accessors and which are mutators, as appropriate.

Exercise 30.2

Change the `main` function provided in the solution to the main activity of this lesson so that it calls a non-member function `inputTransactions` to input all the transactions. Write the `inputTransactions` function so that it contains all the code for displaying the prompt messages as well as the `while` loop and the `switch` statement.

Exercise 30.3

The following class contains a constructor with parameters.

```
class SwimmingPool
{
  public:
    SwimmingPool(int l, int w, int d, int s);
    int volume( ) const;
  private:
    int length, width, deepDepth, shallowDepth;
};

SwimmingPool::SwimmingPool(int l, int w, int d, int s)
{
   length = l;
   width = w;
   deepDepth = d;
   shallowDepth = s;
}
```

Now, when we declare a `SwimmingPool` object, we have to provide values for the four parameters, for example

```
SwimmingPool pool(100, 25, 3, 1);
```

This will set aside memory for a `SwimmingPool` object and initialise the four member variables to the corresponding values.

Write a program to input four integer values and then declare a `SwimmingPool` object with those values. (Note, there is no `inputData` member function. The main function must input the values and then construct an object with those values.)

Then implement the `volume` member function. It must calculate the volume of water in a swimming pool in cubic metres. Assume that all the sides are vertical, but that the bottom of the pool slopes evenly from the shallow end to the deep end. Call this member function in your program and output the volume calculated.

Exercise 30.4

Consider the following interface for a class that plays sound files:

```
class WavSound
{
  public:
    WavSound( );
    void loadFile(string fName);
    bool isLoaded( ) const;
    void play( );
    void stop( );
    void rewind( );
  private:
    bool loaded;
    string fileName;
};
```

Write a program to use this class. Note: You need not write the implementation of the member functions of the `WavSound` class. These have been written by someone else. Your program should simply allow the user to choose a **.wav** file, and then play, stop or rewind the sound file.

An example of the user interaction should look as follows:

```
This program plays .wav sound files
= = = = = = = = = = = = = = = = = =
Choose one of the following options:
(L)oad, (P)lay, (S)top, (R)ewind, e(X)it: P
** You must load a file first **
(L)oad, (P)lay, (S)top, (R)ewind, e(X)it: L
Enter the name of a .wav file: mozart.wav
(L)oad, (P)lay, (S)top, (R)ewind, e(X)it: P
(L)oad, (P)lay, (S)top, (R)ewind, e(X)it: S
(L)oad, (P)lay, (S)top, (R)ewind, e(X)it: R
(L)oad, (P)lay, (S)top, (R)ewind, e(X)it: P
(L)oad, (P)lay, (S)top, (R)ewind, e(X)it: L
Enter the name of a .wav file: beatles.wav
** The file beatles.wav was not found **
(L)oad, (P)lay, (S)top, (R)ewind, e(X)it: X
```

Your program should only declare a single instance of the `WavSound` class. As the user makes each choice, the program should call the appropriate member function of the `WavSound` class.

Note the error messages that your program should display: If the user attempts to play a sound file when a file is not loaded, an error message should be displayed instead of calling the `play` member function. If the user enters a filename for a sound file that the `load` function can't find, or the file is not a valid sound file, `isLoaded` will return `false`. You should use it to warn the user that no file was loaded.

Note: You won't be able to test your program without the implementation of the `WavSound` class. However, with your knowledge of C++ programming you should be able to write the code for a program that would work if you had the implementation. Often in C++ you will be required to write programs that use one or more classes without access to their implementations.

Appendix A

C++ reserved words

```
asm
auto
bool
break
case
catch
char
class
const
const_cast
continue
default
delete
do
double
dynamic_cast
else
enum
explicit
extern
false
float
for
friend
goto
if
inline
int
long
mutable
namespace
new
operator
private
protected
public
register
reinterpret_cast
return
short
signed
sizeof
static
static_cast
struct
switch
template
this
throw
true
try
typedef
typeid
typename
union
unsigned
using
virtual
void
volatile
wchar_t
while
```

Appendix B

Operators

Arithmetic operators

```
+
-
*
/
%
```

(Note: `%` is only used for `int` values)

Boolean operators

```
&&
||
!
```

Relational operators

```
==
<
>
<=
>=
!=
```

Stream operators

```
<<
>>
```

String operators

```
+
```

Operator precedence

The operators below are ordered from top to bottom in decreasing order of precedence. Operators on the same level are evaluated from left to right in an expression.

`( )`					
postfix `++`	postfix `--`				
prefix `++`	prefix `--`	unary `+`	unary `-`	`!`	
`*`	`/`	`%`			
`+`	`-`				
`<<`	`>>`				
`<`	`<=`	`>=`	`>`		
`==`	`!=`				
`&&`					
`||`					
`=`	`*=`	`/=`	`+=`	`-=`	`%=`

Escape characters

`\n`	newline
`\t`	tab
`\v`	vertical tab
`\b`	backspace
`\f`	form feed
`\a`	alert or bell
`\r`	carriage return
`\\`	backslash
`\'`	single quote
`\"`	double quote

Appendix C

ASCII table

0	␀	1	☺	2	☻	3	♥	4	♦	5	♣	6	♠	7	•
8	◘	9	○	10	◙	11	♂	12	♀	13	♪	14	♫	15	☼
16	►	17	◄	18	↕	19	‼	20	¶	21	§	22	▬	23	↨
24	↑	25	↓	26	→	27	←	28	∟	29	↔	30	▲	31	▼
32		33	!	34	"	35	#	36	$	37	%	38	&	39	’
40	(	41	)	42	*	43	+	44	,	45	-	46	.	47	/
48	0	49	1	50	2	51	3	52	4	53	5	54	6	55	7
56	8	57	9	58	:	59	;	60	<	61	=	62	>	63	?
64	@	65	A	66	B	67	C	68	D	69	E	70	F	71	G
72	H	73	I	74	J	75	K	76	L	77	M	78	N	79	O
80	P	81	Q	82	R	83	S	84	T	85	U	86	V	87	W
88	X	89	Y	90	Z	91	[	92	\	93	]	94	^	95	_
96	‘	97	a	98	b	99	c	100	d	101	e	102	f	103	g
104	h	105	i	106	j	107	k	108	l	109	m	110	n	111	o
112	p	113	q	114	r	115	s	116	t	117	u	118	v	119	w
120	x	121	y	122	z	123	{	124	|	125	}	126	~	127	⌂
128	Ç	129	ü	130	é	131	â	132	ä	133	à	134	å	135	ç
136	ê	137	ë	138	è	139	ï	140	î	141	ì	142	Ä	143	Å
144	È	145	æ	146	Æ	147	ô	148	ö	149	ò	150	û	151	ù
152	ÿ	153	Ö	154	Ü	155	¢	156	£	157	¥	158	Pt	159	ƒ
160	á	161	í	162	ó	163	ú	164	ñ	165	Ñ	166	ª	167	º
168	¿	169		170	¬	171	½	172	¼	173	¡	174	«	175	»
176	▒	177	▓	178	▓	179	│	180	┤	181	╡	182	╢	183	╖
184	╕	185	╣	186	║	187	╗	188	╝	189	╜	190	╛	191	┐
192	└	193	┴	194	┬	195	├	196	–	197	┼	198	╞	199	╟
200	╚	201	╔	202	╩	203	╦	204	╠	205	=	206	╬	207	╧
208	╨	209	╤	210	╥	211	╙	212	╘	213	╒	214	╓	215	╫
216	╪	217	┘	218	┌	219	▉	220	▄	221	▌	222	▐	223	▀
224	α	225	ß	226	Γ	227	π	228	Σ	229	σ	230	µ	231	γ
232	Φ	233	θ	234	Ω	235	δ	236	∞	237	$\oint$	238	∈	239	∩
240	≡	241	±	242	≥	243	≤	244		245		246	÷	247	≈
248	°	249	·	250	·	251	√	252	^{n}	253	^{2}	254	■	255	

Appendix D

C++ standard library functions

We only give a selection of the predefined functions in C++. Note that many of these functions are defined in different header files which have to be included to be able to use them.

Void functions

Function signature	Description	Header file
`getline(istream, string &, char)`	Extracts characters from an input stream up to the first occurrence of a delimiting character and copies them to a string	`iostream`
`srand(int)`	Seeds the random number generator with an integer	`cstdlib`

Value-returning functions

Function signature	Description	Header file
`int abs(int)`	absolute value	`cstdlib`
`int ceil(float)`	smallest integer greater than or equal to	`cmath`
`float cos(float)`	cosine	`cmath`
`float exp(float)`	exponential function	`cmath`
`float fabs(float)`	floating point absolute value	`cmath`
`int floor(float)`	largest integer less than or equal to	`cmath`
`float log(float)`	natural logarithm	`cmath`
`float log10(float)`	logarithm base 10	`cmath`
`float pow(float,float)`	power	`cmath`
`int rand( )`	random integer value from 0 to 32767	`cstdlib`
`float sin(float)`	sine	`cmath`
`float sqrt(float)`	square root	`cmath`
`char tolower(char)`	converts to lower case	`iostream`
`char toupper(char)`	converts to upper case	`iostream`
`int time(int)`	current time as a large positive integer	`ctime`

Appendix E

Member functions of the `string` class

The following member functions are provided with the **`string`** class to manipulate the values of string objects:

Accessors

Function signature	Description
`int size( )`	Returns the size (i.e. length) of a string object.
`string substr(int, int)`	Returns a substring of a string object. The first parameter specifies the starting position (i.e. the position from which the substring should be copied) and the second parameter specifies how long the substring should be (i.e. how many characters should be copied). The second parameter may be omitted, in which case the substring consisting of all the characters from the starting position (specified by the first and only parameter) to the end of the string are returned.
`int find(string, int)`	Returns the position of a string (specified as the first parameter) within a string object. The second parameter is optional, and can be used to specify where the search has to be commenced. If omitted, the search commences at the beginning of the string object. If the string being sought is not found, -1 is returned.

Mutators

Function signature	Description
`void insert(int, string)`	Inserts a string (specified as the second parameter) into a string object at a particular position (specified as the first parameter)
`void erase(int, int)`	Erases a substring from a string object. The substring that is to be erased is determined by the two parameters: from the position specified by the first parameter, as many characters as specified by the second parameter
`void replace(int, int, string)`	Replaces specified characters of a string object with another string. The characters to be replaced are determined by the first two parameters: from the position specified by the first parameter, as many characters as specified by the second parameter. The string to be inserted in their place is specified by the third parameter.