# Part IA Computing Course
# Tutorial Guide to C++ Programming

Roberto Cipolla
Department of Engineering
University of Cambridge

September 1, 2004

This document provides an introduction to computing and the C++ programming language. It will teach you how to write, compile, execute and test simple computer programs in C++ and describes the computing exercises to be completed in the Michaelmas term. This tutorial guide and the *Introduction for New Users* should be read before the first laboratory session. An outline of the computing course including details of the objectives, organisation and timetable of the laboratory sessions can be found on pages 4–5.

# Contents

# Part IA Computing Course Outline

## A. Aims of Michaelmas Term Computing Course

This guide provides a tutorial introduction to computing and the C++ computer programming language. It will teach you how to **write, compile, execute and test simple computer programs** in C++ which read data from the keyboard, perform some computations on that data, and finally print out the results or display them graphically.

## B. Objectives

### Session 1

☐ Familiarisation with the teaching system and C++ development environment
☐ Declaration and definition of variables and constants
☐ Assignment of values to variables and their manipulation in arithmetic expressions
☐ Writing the value of variables to the screen
☐ Reading the value of variables from keyboard
☐ Editing, compiling, debugging and executing a simple program

### Session 2

☐ Boolean expressions and relational operators
☐ Simple control structures for selection and repetition

### Session 3

☐ Definition, declaration and calling of functions
☐ Passing values to and returning values from functions
☐ Math and system library functions

### Session 4

☐ Array data structure
☐ Declaration and initialisation of arrays
☐ Passing arrays to functions
☐ Graphics library functions

## C. Organisation and Marking

There are 4 two hour laboratory sessions scheduled for the Michaelmas term computing course. Part of the first session should be spent working through the *Introduction for New Users* which will teach you how to use the teaching computer system. You then have the remaining three and a half sessions in which to work through the examples and exercises described in this document. Each session consists of a tutorial (which must be read **before** the start of the laboratory session) and computing exercises.

> **(0)** The computing exercises are placed inside a numbered box like this one. They should be completed before moving on to the next session. Working programs should be shown to a demonstrator and will be marked in order to qualify for the 12 coursework marks available for this course. This should be done by the end of each day (i.e. session 2 and session 4). Exercises marked with a (*****) are optional but should be attempted if time permits.

## D. Timetable

The computing course requires approximately **8 hours** in timetabled laboratory sessions with an additional **2 to 4 hours** for reading the tutorial guide and preparation **before** the laboratory sessions.

| Session number | Objectives and exercises | Time required |
|---|---|---|
| (Preparation) | Read tutorial sections 1–6 before session 1 | 90 minutes |
| Session 1 | Introductory lecture | 30 minutes |
| (Day 1 - morning) | Familiarisation with teaching system | 30 minutes |
| | Computing exercises (1), (2) and (3) | 60 minutes |
| Session 2 | Computing exercises (4) and (5) | 60 – 120 minutes |
| (Day 1 - afternoon) | Marking | |
| (Preparation) | Read tutorial sections 7–10 before session 3 | 120 minutes |
| Session 3 | Computing exercises (6) and (7*) | 60 – 120 minutes |
| (Day 2 - morning) | | |
| Session 4 | Computing exercises (8), (9) and (10*) | 60 – 120 minutes |
| (Day 2 - afternoon) | Marking | |
| | Read tutorial section 11 | 30 minutes |

To benefit most from the laboratory session and help from the demonstrators you must **read the tutorial sections before the start of each session**. In particular, at the start of session 3, you should be ready to test the sample programs described in tutorial sections 7–8 and start the computing exercises.

# 1 Introduction

## 1.1 What is a computer program?

Computers process data, perform computations, make decisions and instigate actions under the control of sets of instructions called **computer programs**. The computer (central processing unit, keyboard, screen, memory, disc) is commonly referred to as the **hardware** while the programs that run on the computer (including operating systems, word processors and spreadsheets) are referred to as **software**.

Program instructions are stored and processed by the computer as a sequence of binary digits (i.e. 1's and 0's) called *machine code*. In the early days programmers wrote their instructions in strings of numbers called **machine language**. Although these machine instructions could be directly read and executed by the computer, they were too cumbersome for humans to read and write. Later *assemblers* were developed to map machine instructions to English-like abbreviations called mnemonics (or **assembly language**). In time, **high-level programming languages** (e.g. FORTRAN (1954), COBOL (1959), BASIC (1963) and PASCAL (1971)) developed which enable people to work with something closer to the words and sentences of everyday language. The instructions written in the high-level language are automatically translated by a **compiler** (which is just another program) into binary machine instructions which can be executed by the computer at a later stage.

**Note** that the word "program" is used to describe both the set of written instructions created by the programmer and also to describe the entire piece of executable software.

## 1.2 The C++ Programming Language

The C++ programming language (Stroustrup (1988)) evolved from C (Ritchie (1972)) and is emerging as the standard in software development. For example, the *Unix* and *Windows* operating systems and applications are written in C and C++. It facilitates a structured and disciplined approach to computer programming called *object-oriented programming*. It is a very powerful language and in its full form it is also a rich language. However, in this course, only the basic elements of C++ will be covered.

# 2 Editing, Compiling and Executing a Simple Program

## 2.1 A simple C++ program to add two numbers

The following is an example of a simple program (source code) written in the C++ programming language. The program is short but nevertheless complete. The program is designed to read two numbers typed by a user at the keyboard; compute their sum and display the result on the screen.

```
// Program to add two integers typed by user at keyboard
#include <iostream>
using namespace std;

int main()
{
   int a, b, total;

   cout << "Enter integers to be added:" << endl;
   cin >> a >> b;
   total = a + b;
   cout << "The sum is " << total << endl;

   return 0;
}
```

## 2.2 Overview of program structure and syntax

C++ uses notation that may appear strange to non-programmers. The notation is part of the **syntax** of a programming language, i.e. the formal rules that specify the structure of a legal program. The notation and explanations which follow will appear strange if you have never written a computer program. Do not worry about them or how the program works. This will be explained in more detail later, in section 3, 4 and 5. The following is an overview.

Every C++ program consists of a header and a main body and has the following structure.

```
// Comment statements which are ignored by computer but inform reader
#include < header file name>

int main()
{
    declaration of variables;
    statements;

    return 0;
}
```

We will consider each line of the program: for convenience the program is reproduced on the next page with line numbers to allow us to comment on details of the program. You must **not** put line numbers in an actual program.

```
1  // Program to add two integers typed by user at keyboard
2  #include <iostream>
3  using namespace std;
4
5  int main()
6  {
7     int a, b, total;
8
9     cout << "Enter integers to be added:" << endl;
10    cin >> a >> b;
11    total = a + b;
12    cout << "The sum is " << total << endl;
13
14    return 0;
15 }
```

**Line 1** At the top of the program are comments and instructions which will not be executed. Lines beginning with `//` indicate that the rest of the line is a **comment**. Comments are inserted by programmers to help people read and understand the program. Here the comment states the purpose of the program. Comments are not executed by the computer. They can in general be placed anywhere in a program.

**Line 2** Lines beginning with `#` are instructions to the compiler's *preprocessor*. The **include** instruction says "what follows is a file name, find that file and insert its contents right here". It is used to include the contents of a file of definitions which will be used in the program. Here the file `iostream` contains the definitions of some of the symbols used later in the program (e.g. **cin**, **cout**).

**Line 3** This is an advanced feature of C++. It is used to specify that names used in the program (such as **cin** and **cout**) are defined in the standard C and C++ libraries. This is used to avoid problems with other libraries which may also use these names.

**Line 5** When the program is executed the instructions will be executed in the order they appear in the main body of the program. The main body is delimited by `main()` and the opening and closing braces (curly brackets). This line also specifies that `main()` will return a value of type integer (`int`) on its completion (see line 14). Every C++ program, irrespective of what it is computing, begins in this way.

**Line 6** The opening (left) brace marks the beginning of the main body of the program. The main body consists of instructions which are **declarations** defining the data or **statements** on how the data should be processed. All C++ declarations and statements must end with a semicolon.

**Line 7** This is a declaration. The words `a`, `b` and `total` are the names of **variables**. A variable is a location in the computer's memory where a value can be stored for use by a program. We can assign and refer to values stored at these locations by simply using the variable's name. The declaration also specifies the variable **type**. Here

the variables `a`, `b` and `total` are declared to be data of type **int** which means these variables hold integer values. At this stage, the values of the variables are undefined.

**Line 9** This statement instructs the computer to output the *string* of characters contained between the quotation marks, followed by a new line (`endl`). The location of the output is denoted by `cout` which in this case will be the terminal screen.

**Line 10** This statement instructs the computer to read data typed in at the keyboard (standard input), denoted by **cin**. These values are *assigned to* (stored in) variables `a` and `b`.

**Line 11** This statement is an **arithmetic expression** which assigns the value of the expression `a + b` (the sum of the integer values stored at `a` and `b`) to the variable `total`.

**Line 12** Instructs the computer to display the value of the variable `total`.

**Line 14** The last instruction of every program is the `return` statement. The return statement with the integer value 0 (zero) is used to indicate to the operating system that the program has terminated successfully.

**Line 15** The closing (right) brace marks the end of the main body of the program.

**Blank lines** (Lines 4, 8 and 13) have been introduced to make the program more readable. They will be ignored by the compiler. **Whitespace** (spaces, tabs and newlines) are also ignored (unless they are part of a string of characters contained between quotation marks). They too can be used to enhance the visual appearance of a program.

**Indentation** It does not matter where you place statements, either on the same line or on separate lines. A common and accepted *style* is that you indent after each opening brace and move back at each closing brace.

## 2.3 The development environment and the development cycle

C++ programs go through 3 main phases during development: editing (writing the program), compiling (i.e. translating the program to executable code and detecting syntax errors) and running the program and checking for logical errors (called *debugging*).

1. **Edit**
   The first phase consists of editing a file by typing in the C++ program with a text editor and making corrections if necessary. The program is stored as a text file on the disk, usually with the file extension `.cc` to indicate that it is a C++ program (e.g. `SimpleAdder.cc` ).

2. **Compile**
   A compiler translates the C++ program into machine language code (**object code**) which it stores on the disk as a file with the extension `.o` ( e.g. `SimpleAdder.o`). A linker then links the object code with standard library routines that the program may use and creates an **executable image** which is also saved on disk, usually as a file with the file name without any extension (e.g. `SimpleAdder`).

3. **Execute**
   The executable is loaded from the disk to memory and the computer's processing unit (Central Processing Unit) executes the program one instruction at a time.

# Part IA Computing Course — Session 1A

## A. Objectives

☐ Familiarisation with the teaching system and C++ development environment
☐ Edit, compile and execute a working program

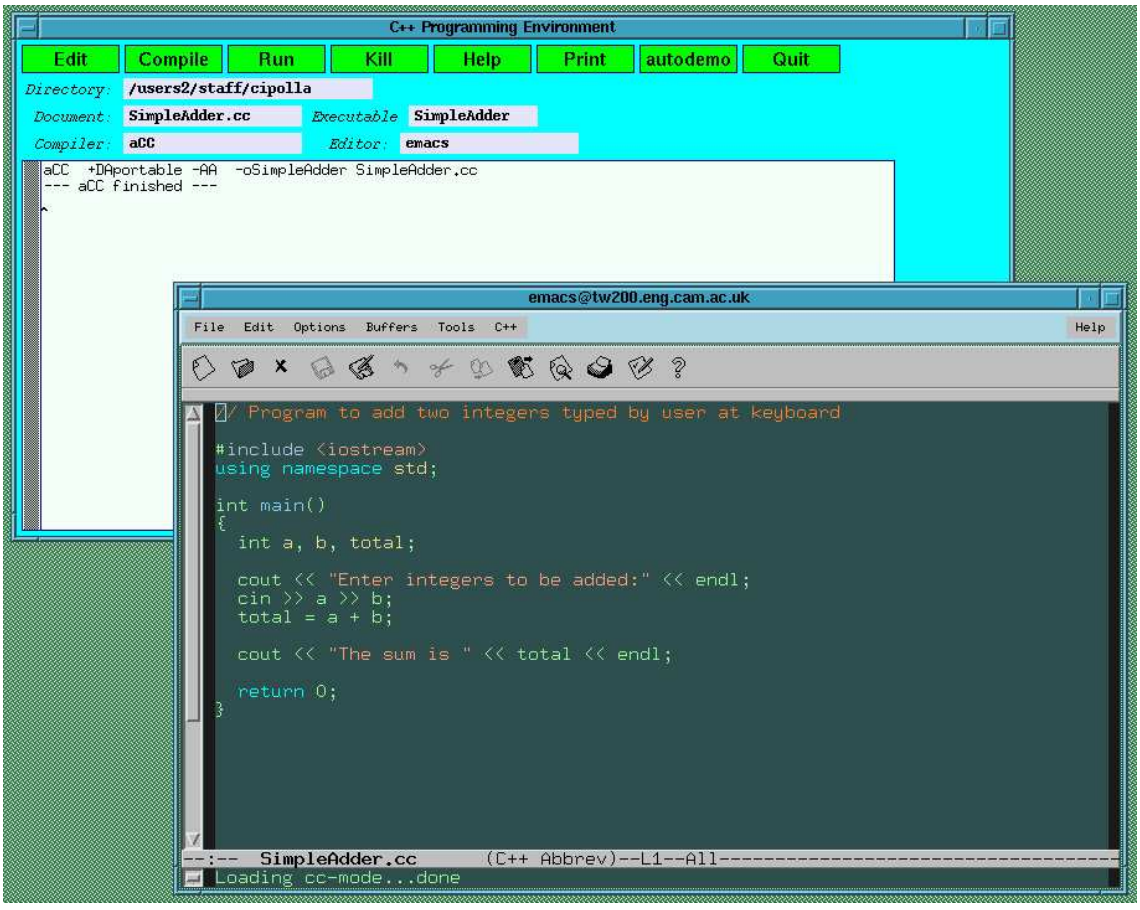
## B. Getting started

If you have not already done so, read sections 1–2 of the tutorial guide and then log onto a computer in the CUED teaching system by typing in your user identifier and password. Start the File Manager environment by typing:

```
start 1AComputing
```

This command will set up the C++ Programming Environment (`xcc window`) which will be used for creating and executing the software developed in this laboratory.

We will start with the short, but nevertheless complete, C++ program to add two integers, shown in the figure below.



The program is the basis of the first computing exercise, (1), described in section C. Detailed instructions on how to enter the program, compile it and then get the computer to execute the instructions using the C++ Programming Environment are given in section D.

## C. Computing Exercise

**(1)** Use an editor to type in the simple adder program (`SimpleAdder.cc`) listed above and then compile and run it.

Execute it with different numbers at the input and check that it adds up the numbers correctly. What happens if you input two numbers with decimal places?

## D. Editing, compiling and executing the simple adder program

1. **Edit**

   Create a new file by typing in the name of the file in the box titled `document` in the C++ Programming Environment (`xcc`) window. Give the file the name `SimpleAdder.cc` – the `.cc` extension informs the editor that the file will be a C++ program. Avoid putting spaces into the name of the file name.

   Select the `Edit` button of the (`xcc`) window by a single click with the left mouse button. This will start up an **Emacs** text editor window in which to edit your file. When the editor is ready, type in the `SimpleAdder.cc` program exactly as shown above. Be particularly careful to get all the punctuation correct, including the right brace mark at the end followed by a new line. Make sure to include the `return 0` (numeric 0). To save the edited file select the `Save Buffer` option on the emacs editor `Files` menu.

2. **Compile**

   To compile your program click on the `Compile` button. The ANSI C++ `aCC` compiler will attempt to compile the program and link it with any other routines that the program may need. Provided you have not made any typing mistakes the program will be successfully compiled (although this may take some time) and the message `--- aCC finished ---` will appear in the `xcc` window. An executable called `SimpleAdder` will have been created in your directory.

   If you have made a typing mistake, find the error and correct it using the editor. Save the corrected file using the `Save Buffer` option and try the compilation again.

3. **Run**

   To execute the program click on the `Run` button. When the program prompt appears, input the two integer numbers to the program by typing in two numbers separated by spaces and followed by a return. The program should respond by printing out the sum of the two numbers.

   You can run the program as many times as you wish. If you accidently select the `Quit` button you can restart the C++ Programming Environment by *double clicking* on the C++ Compiler icon in the **Applications window**. Another way of modifying,

compiling or executing an existing file is to double click on the program file icon (e.g. `SimpleAdder.cc`) in the **File Manager** window. This will start the C++ Programming Environment with the file name already typed in for you.

### E. Animation of the execution of the SimpleAdder.cc program

An animation of the execution of the `SimpleAdder.cc` program to help visualise the role of the Central Processing Unit, program and data memory and input and output units in the execution the C++ program can be seen by clicking on the `Help` button and selecting the Tutorial Guide animation on the *web* help page that appears.

# 3 Variables and Constants

Programs need a way to store the data they use. Variables and constants offer various ways to represent and manipulate data. Constants, as the name suggests, have fixed values. Variables, on the other hand, hold values which can be assigned and changed as the program executes.

## 3.1 Variable types

Every variable and constant has an associated **type** which defines the set of values that can be legally stored in it. Variables can be conveniently divided into integer, floating point, character and boolean types for representing integer (whole) numbers, floating point numbers (real numbers with a decimal point), the ASCII character set (for example 'a', 'b', 'A') and the boolean set (`true` or `false`) respectively.

More complicated types of variable can be defined by a programmer, but for the moment, we will deal with just the simple C++ types. These are listed below:

| | |
|---|---|
| **int** | to store a positive or negative integer (whole) number |
| **float** | to store a real (floating point) number |
| **bool** | to store the logical values `true` or `false` |
| **char** | to store one of 256 character (text) values |

## 3.2 Declaration of a variable

A variable is introduced into a program by a declaration which states its **type** (i.e. `int`, `float` , `bool` or `char`) and its name, which you are free to choose. A **declaration** must take the form:

*type* *variable-name*;

```
int           count;
float         length;
char          firstInitial;
bool          switched_on;
```

or:

*type* *variable1*, *variable2*, ... *variableN*;

```
float         base, height, areaCircle;
int           myAge, number_throws;
```

The variable name can be any sequence of characters consisting of letters, digits and underscores that do not begin with a digit. It must not be a special keyword of the C++ language and cannot contain spaces. C++ is case-sensitive: uppercase and lowercase letters are considered to be different. Good variable names tell you how the variable is used and help you understand the flow of the program. Some names require two words and this can be indicated by using the underscore symbol (`_`) or using an uppercase letter for the beginning of words.

### 3.3 Storage of variables in computer memory

When you run your program it is loaded into computer memory (RAM) from the disk file. A variable is in fact a location in the computer's memory in which a value can be stored and later retrieved. The variable's name is merely a label for that location – a *memory address*. It may help to think of variables as named boxes into which values can be stored and retrieved.

The amount of memory required for the variables depends on their type. This can vary between machines and systems but is usually one byte (8 bits) for a `char` variable, four bytes for an `int` and four bytes for a `float`. This imposes limits on the range of numbers assigned to each variable. Integer numbers must have values in the range −2147483648 to 2147483647 (i.e. $\pm 2^{31}$). Floats must be real numbers with magnitudes in the range $5.9 \times 10^{-39}$ to $3.4 \times 10^{38}$ (i.e. $2^{-127}$ to $2^{128}$). They are usually stored using 1 bit for the sign (s), 8 bits for the exponent (e) and 23 bits for the mantissa (m) such that the number is equal to $s \times m \times 2^e$. The ratio of the smallest and largest numbers that can be correctly added together must therefore be greater than $2^{-23} \approx 10^{-7}$ (i.e. 7 digits of accuracy). This depends only on the number of bits used to represent the mantissa.

If an application requires very small or large numbers beyond these ranges, C++ offers two additional data types for integers and floating point numbers: `long` and `double`. Variables of type `double` require double the amount of memory for storage but are useful when computing values to a high precision.

## 4 Assignment of variables

### 4.1 Assignment statements

It is *essential* that every variable in a program is given a value explicitly before any attempt is made to use it. It is also very important that the value assigned is of the correct type.

The most common form of statement in a program uses the **assignment operator**, `=`, and either an **expression** or a **constant** to assign a value to a variable:

*variable* `=` *expression*`;`
*variable* `=` *constant*`;`

The symbol of the assignment operator looks like the mathematical equality operator **but** in C++ its meaning is different. The assignment statement indicates that the value given by the expression on the right hand side of the **assignment operator** (symbol `=`) must be stored in the variable named on the left hand side. The assignment operator should be read as "becomes equal to" and means that the variable on the left hand side has its value changed to the value of the expression on the right hand side. For the assignment to work successfully, the type of the variable on the left hand side should be the same as the type returned by the expression.

The statement in line 10 of the simple adder program is an example of an assignment statement involving an **arithmetic expression**.

```
total = a + b;
```

It takes the values of `a` and `b`, sums them together and assigns the result to the variable `total`. As discussed above, variables can be thought of as named boxes into which values

can be stored. Whenever the name of a box (i.e. a variable) appears in an expression, it represents the value currently stored in that box. When an assignment statement is executed, a new value is dropped into the box, replacing the old one. Thus, line 10 of the program means "get the value stored in the box named `a`, add it to the value stored in the box named `b` and store the result in the box named `total`".

The assignment statement:

```
total = total + 5;
```

is thus a valid statement since the new value of `total` becomes the old value of `total` with 5 added to it. Remember the assignment operator (`=`) is **not** the same as the equality operator in mathematics (represented in C++ by the operator `==`).

## 4.2 Arithmetic expressions

Expressions can be constructed out of variables, constants, operators and brackets. The commonly used mathematical or **arithmetic operators** include:

| operator | operation |
|---|---|
| `+` | addition |
| `-` | subtraction |
| `*` | multiplication |
| `/` | division |
| `%` | modulus (modulo division) |

The definitions of the first four operators are as expected. The modulo division (modulus) operation with an integer is the remainder after division, e.g. 13 modulus 4 (`13%4`) gives the result 1. Obviously it makes no sense at all to use this operator with `float` variables and the compiler will issue a warning message if you attempt to do so.

Although addition, subtraction and multiplication are the same for both integers and reals (floating point numbers), division is different. If you write (see later for declaration and initialisation of variables on the same line):

```
float a=13.0, b=4.0, result;
result = a/b;
```

then a real division is performed and 3.25 is assigned to `result`. A different result would have been obtained if the variables had been defined as integers:

```
int i=13,j=4, result;
result = i/j;
```

when `result` is assigned the integer value 3.

The remainder after integer division can be determined by the modulo division (modulus) operator, `%`. For example, the value of `i%j` would be 1.

## 4.3 Precedence and nesting parentheses

The use of parentheses (brackets) is advisable to ensure the correct evaluation of complex expressions. Here are some examples:

| | |
|---|---|
| `4 + 2 * 3` | equals 10 |
| `(4+2) * 3` | equals 18 |
| `-3 * 4` | equals -12 |
| `4 * -3` | equals -12 (but should be avoided) |
| `4 * (-3)` | equals -12 |
| `0.5(a+b)` | illegal (missing multiplication operator) |
| `(a+b) / 2` | equals the average value of a and b only if they are of type float |

The order of execution of mathematical operations is governed by rules of precedence. These are similar to those of algebraic expressions. Parentheses are always evaluated first, followed by multiplication, division and modulus operations. Addition and subtraction are last. The best thing, however, is to use parentheses (brackets) instead of trying to remember the rules.

## 4.4 Initialisation of variables

Variables can be assigned values when they are first defined (called initialisation):

*type* *variable* = *literal constant*;

```
float   ratio = 0.8660254;
int     myAge = 19;
char    answer = 'y';
bool    raining = false;
```

The terms on the right hand side are called constants [1].

The declaration of a variable and the assignment of its value in the same statement can be used to define variables as they are needed in the program.

*type* *variable* = *expression*;

```
float   product = factor1*factor2;
```

The variables in the expression on the right hand side must of course have already been declared and had values assigned to them [2].

[1] Note the ASCII character set is represented by type `char`. Each character constant is specified by enclosing it between single quotes (to distinguish it from a variable name). Each `char` variable can only be assigned a single character. These are stored as numeric codes. (The initialisation of words and character strings will be discussed later in the section on advanced topics.)

[2] **Warning:** When declaring and initialising variables in the middle of a program, the variable exists (i.e. memory is assigned to store values of the variable) up to the first right brace that is encountered, excluding any intermediate nested braces, {}. For the simple programs described here, this will usually be the closing brace mark of the program. However we will see later that brace marks can be introduced in many parts of the program to make compound statements.

## 4.5 Expressions with mixed variable types

At a low level, a computer is not able to perform an arithmetic operation on two different data types of data. In general, only variables and constants of the *same* type, should be combined in an expression. The compiler has strict *type checking* rules to check for this.

In cases where mixed numeric types appear in an expression, the compiler replaces all variables with copies of the highest precision type. It *promotes* them so that in an expression with integers and float variables, the integer is automatically converted to the equivalent floating point number for the purpose of the calculation only. The value of the integer is not changed in memory. Hence, the following is legal:

```
int i=13;
float x=1.5;
x = (x * i) + 23;
```

since the values of `i` and `23` are automatically converted to floating point numbers and the result is assigned to the float variable `x`. However the expression:

```
int i=13,j=4;
float result;
result = i/j;
```

is evaluated by integer division and therefore produces the incorrect assignment of `3.0` for the value of `result`. You should try and avoid expressions of this type but occasionally you will need to compute a fraction from integer numbers. In these cases the compiler needs to be told specifically to convert the variables on the right-hand side of the assignment operator to type `float`. This is done by **casting**.

In the C++ language this is done by using the construction:[3]

`static_cast<` *type* `>` *expression*

For example:

```
int count=3, N=100;
float fraction;
fraction  = static_cast<float>(count)/N;
```

converts (casts) the value stored in the integer variable `count` into a floating point number, 3.0. The integer `N` is then promoted into a floating point number to give a floating point result.

## 4.6 Declaration and initialisation of symbolic constants

Like variables, **symbolic** constants have types and names. A constant is declared and initialised in a similar way to variables **but** with a specific instruction to the compiler that the value cannot be changed by the program. The values of constants must always be assigned when they are created.

[3] In the C language this is done by a different construction using: (type) expression.

```
const   type       constant-name =  literal constant;

const   float    Pi = 3.14159265;
const   int      MAX = 10000;
```

The use of constants helps programmers avoid inadvertent alterations of information that should never be changed. The use of appropriate constant names instead of using the numbers also helps to make programs more readable.

# 5 Simple Input and Output

C++ does not, as part of the language, define how data is written to a screen, nor how data is read into a program. This is usually done by "special variables" (*objects*) called **input and output streams**, `cin` and `cout`, and the **insertion** and **extraction operators**. These are defined in the **header file** called `iostream`. To be able to use these *objects* and operators you must include the file `iostream` at the top of your program by including the following line of code before the main body in your program.

```
#include <iostream>
```

## 5.1 Printing to the screen using output stream

A statement to print the value of a variable or a **string of characters** (set of characters enclosed by double quotes) to the screen begins with `cout`, followed by the **insertion operator**, (`<<`) which is created by typing the "less than" character (`<`) twice. The data to be printed follows the insertion operator.

```
cout << " text to be printed ";
cout <<  variable;
cout <<  endl;
```

The symbol `endl` is called a **stream manipulator** and moves the cursor to a new line. It is an abbreviation for *end of line*.

Strings of characters and the values of variables can be printed on the same line by the repeated use of the insertion operator. For example (line 11 of the simple adder program):

```
int total = 12;
cout << "The sum is " << total << endl;
```

prints out

```
The sum is 12
```

and then moves the cursor to a new line.

## 5.2 Input of data from the keyboard using input stream

The input stream *object* `cin` and the **extraction operator**, (`>>`), are used for reading data from the keyboard and assigning it to variables.

`cin >>` *variable* `;`
`cin >>` *variable1* `>>` *variable2* `;`

Data typed at the keyboard followed by the *return* or *enter* key is assigned to the variable. The value of more than one variable can be entered by typing their values on the same line, separated by spaces and followed by a return, or on separate lines.

# Part IA Computing Course — Session 1B

## A. Objectives

After reading through sections 3 to 5 of the tutorial guide and working through the examples you should be able to:

☐ Declare and define variables and constants
☐ Assign values to variables and manipulate them in arithmetic expressions
☐ Write the value of variables to the screen
☐ Read in the value of variables from the keyboard

You will now use these skills to write and execute the programs described in the following computing exercises (section B). Details and hints on the writing and *debugging* of the programs are given in section C.

## B. Computing Exercises

**(2)** Modify the `SimpleAdder.cc` program so that it performs the division of the two integer numbers input by the user and displays the result as a floating point number.

**(3)** Write and test a program to calculate the area of a circle.

The value of $\pi$ should be declared as a constant equal to 3.14159.
The user is to be prompted to type in the radius at the keyboard and a sentence giving the values of the area and radius should then be displayed on the screen.

How can you test that your program is producing the correct results? Try running it with radii of 0, 1 and 2.

## C. Debugging the programs of exercises (2) and (3)

1. **Start with a working program**

   The easiest way to write the programs for the problems is to modify the existing adder program. Save the emacs buffer containing the `SimpleAdder.cc` file using the `Save Buffer As...` option on the editor `Files` menu. You must give it a new name, for example `RealDivision.cc`. Edit this new file with the emacs editor.

2. **Making modifications**

   Have a go at modifying the program to solve the problem.

   - Mentally check that the program is doing what it is required to do.
   - Check that your program is laid out correctly with *indentation* at the beginning and *semi-colons* at the end of each statement.

- Include comment statements to help you and others understand what the program is doing.

When you have finished the new program save the file to disk using the `Save Buffer` option on the editor `Files` menu.

3. **Checking syntax**

   Change the document name in the C++ Environment window to `RealDivision.cc` and select the `Compile` button. If there are no syntax errors, the program will compile successfully. One or more errors will cause the compiler to print out error or warning messages which will be listed along with the program line numbers where the error occurred.

4. **Compilation errors**

   Don't be worried by the number of error messages. The compiler is trying to give as much help as it can. You can get a lot of error messages from just one mistake so just look at the first error message.

   Find the line in the file to which it refers (the `Goto-line` option on the editor `Edit` menu can be used to locate the line quickly) and try to correct it. Often, the compiler will report that the error is in a line just *after* the one in which there was a mistake. If you cannot spot the mistake straight away, look at neighbouring lines.

   The most common errors will be due to undeclared variables or variables incorrectly declared, and missing or incorrect punctuation. Don't spend a long time agonising over the cause of an error message — ask a demonstrator for help.

5. **Debugging**

   When you have corrected the error associated with the first error message you can have a look at the second error message. If it looks very easy to fix you can have a go at fixing it. Save the file `RealDivision.cc` to disk using the `Save Buffer` option and try the compilation again. Click on the `Compile` button in the C++ Environment window. You don't need to close the editor window each time and you should only have one copy of Emacs running at any time.

   Continue with this compile/debug/edit procedure until your program compiles successfully.

6. **Finding logical errors**

   Run your program and check that it gives the correct answer.

# 6 Control Statements

The statements in the programs presented above have all been sequential, executed in the order they appear in the main program.

In many programs the values of variables need to be tested, and depending on the result, different statements need to be executed. This facility can be used to **select** among alternative courses of action. It can also be used to build **loops** for the **repetition** of basic actions.

## 6.1 Boolean expressions and relational operators

In C++ the testing of *conditions* is done with the use of **Boolean expressions** which yield `bool` values that are either `true` or `false`. The simplest and most common way to construct such an expression is to use the so-called **relational operators**.

| | |
|---|---|
| `x == y` | true if `x` is equal to `y` |
| `x != y` | true if `x` is not equal to `y` |
| `x > y` | true if `x` is greater than `y` |
| `x >= y` | true if `x` is greater than or equal to `y` |
| `x < y` | true if `x` is less than `y` |
| `x <= y` | true if `x` is less than or equal to `y` |

Be careful to avoid mixed-type comparisons. If `x` is a floating point number and `y` is an integer the equality tests will **not** work as expected.

## 6.2 Compound boolean expressions using logical operators

If you need to test more than one relational expression at a time, it is possible to combine the relational expressions using the **logical operators**.

| operator | C++ symbol | example |
|---|---|---|
| AND | && | expression1 && expression2 |
| OR | \|\| | expression1 \|\| expression2 |
| NOT | ! | !expression |

The meaning of these will be illustrated in examples below.

## 6.3 The IF selection control statement

The simplest and most common selection structure is the `if` statement which is written in a statement of the form:

`if(` *boolean-expression* `)` *statement*`;`

The **if** statement tests for a particular condition (expressed as a boolean expression) and only executes the following statement(s) if the condition is true. An example follows of a fragment of a program which tests if the denominator is not zero before attempting to calculate `fraction`.

```
if(total != 0)
   fraction = counter/total;
```

If the value of `total` is `0`, the boolean expression above is false and the statement assigning the value of `fraction` is ignored.

If a sequence of statements is to be executed, this can be done by making a **compound statement** or **block** by enclosing the group of statements in braces.

```
if( boolean-expression )
{
     statements;
}
```

An example of this is:

```
if(total != 0)
{
   fraction = counter/total;
   cout << "Proportion = " << fraction <<  endl;
}
```

## 6.4 The IF/ELSE selection control statement

Often it is desirable for a program to take one branch if the condition is true and another if it is false. This can be done by using an **if/else** selection statement:

```
if( boolean-expression )
     statement-1;
else
     statement-2;
```

Again, if a sequence of statements is to be executed, this is done by making a compound statement by using braces to enclose the sequence:

```
if( boolean-expression )
{
     statements;
}
else
{
     statements;
}
```

An example occurs in the following fragment of a program to calculate the roots of a quadratic equation.

```
// testing for real solutions to a quadratic
d = b*b - 4*a*c;
if(d >= 0.0)
{
   // real solutions
   root1 = (-b + sqrt(d)) / (2.0*a);
   root2 = (-b - sqrt(d)) / (2.0*a);
   real_roots = true;
}
else
{
   // complex solutions
   real = -b / (2.0*a);
   imaginary = sqrt(-d) / (2.0*a);
   real_roots = false;
}
```

If the boolean condition is `true`, i.e. (d $\geq$ 0), the program calculates the roots of the quadratic as two real numbers. If the boolean condition tests `false`, then a different sequence of statements is executed to calculate the real and imaginary parts of the complex roots. Note that the variable `real_roots` is of type `bool`. It is assigned the value `true` in one of the branches and `false` in the other.

### 6.5 ELSE IF multiple selection statement

Occasionally a decision has to be made on the value of a variable which has more than two possibilites. This can be done by placing `if` statements within other `if-else` constructions. This is commonly known as *nesting* and a different style of indentation is used to make the multiple-selection functionality much clearer. This is given below:

```
if( boolean-expression-1 )
     statement-1;
else if( boolean-expression-2 )
     statement-2;
⋮
else
     statement-N;
```

For compound statement blocks braces must be used.

## 6.6 SWITCH multiple selection statement

Instead of using multiple `if/else` statements C++ also provides a special control structure, `switch`.

For a variable `x` the `switch(x)` statement tests whether `x` is equal to the constant values `x1`, `x2`, `x3`, etc. and takes appropriate action. The `default` option is the action to be taken if the variable does not have any of the values listed.

```
switch( x )
{
   case  x1:
       statements1;
       break;

   case  x2:
       statements2;
       break;

   case  x3:
       statements3;
       break;

   default:
       statements4;
       break;
}
```

The `break` statement causes the program to proceed to the first statement after the `switch` structure. Note that the `switch` control structure is different to the others in that braces are not required around multiple statements.

The following example uses the `switch` statement to produce a simple calculator which branches depending on the value of the operator being typed in. The operator is read and stored as a character value (`char`). The values of `char` variables are specified by enclosing them between single quotes. The program is terminated (`return -1`) if two numbers are not input or the simple arithmetic operator is not legal. The return value of -1 instead of 0 signals that an error took place [4].

[4] A copy of all the programs listed in this document can be found in the `1AC++Examples` folder in your directory. You should compile and run these programs to help get a better understanding of how they work. Change directory and *double click* on the icon with the file name `CalculatorSwitch.cc`. Compile and run the program.

```
// CalculatorSwitch.cc
// Simple arithmetic calculator using switch() selection.

#include <iostream>
using namespace std;

int main()
{
   float a, b, result;
   char operation;

   // Get numbers and mathematical operator from user input
   cin >> a >> operation >> b;

   // Character constants are enclosed in single quotes
   switch(operation)
   {
   case '+':
         result = a + b;
         break;

   case '-':
         result = a - b;
         break;

   case '*':
         result = a * b;
         break;

   case '/':
         result = a / b;
         break;

   default:
         cout << "Invalid operation. Program terminated." << endl;
         return -1;
   }

   // Output result
   cout << result << endl;
   return 0;
}
```

## 6.7 The WHILE repetition control statement

Repetition control statements allow the programmer to specify actions which are to be repeated while some condition is `true`. In the `while` repetition control structure:

```
while( boolean-expression )
{
    statements;
}
```

the boolean expression (condition) is tested and the statements (or statement) enclosed by the braces are (is) executed repeatedly while the condition given by the boolean expression is true. The loop terminates as soon as the boolean expression is evaluated and tests false. Execution will then continue on the first line after the closing brace.

Note that if the boolean expression is initially false the statements (or statement) are not executed. In the following example the boolean condition becomes false when the first negative number is input at the keyboard. The sum is then printed. [5]

```
// AddWhilePositive.cc
// Computes the sum of numbers input at the keyboard.
// The input is terminated when input number is negative.

#include <iostream>
using namespace std;

int main()
{
   float number, total=0.0;

   cout << "Input numbers to be added: " << endl;
   cin >> number;

   // Stay in loop while input number is positive
   while(number >= 0.0)
   {
       total = total + number;
       cin >> number;
   }

   // Output sum of numbers
   cout << total << endl;
   return 0;
}
```

[5] *Double click* on the icon with the file name `AddWhilePositive.cc` and compile and run the program.

## 6.8 Increment and decrement operators

Increasing and decreasing the value of an integer variable is a commonly used method for counting the number of times a loop is executed. C++ provides a special operator `++` to increase the value of a variable by 1. The following are equivalent ways of *incrementing* a counter variable by 1.

```
count = count + 1;
count++;
```

The operator `--` decreases the value of a variable by 1. The following are both *decrementing* the counter variable by 1.

```
count = count - 1;
count--;
```

## 6.9 The FOR repetition control statement

Often in programs we know how many times we will need to repeat a loop. A `while` loop could be used for this purpose by setting up a starting condition; checking that a condition is true and then incrementing or decrementing a counter within the body of the loop. For example we can adapt the `while` loop in `AddWhilePositive.cc` so that it executes the loop `N` times and hence sums the `N` numbers typed in at the keyboard.

```
i=0;          // initialise counter
while(i<N)    // test whether counter is still less than N
{
   cin >> number;
   total = total + number;
   i++;       // increment counter
}
```

The initialisation, test and increment operations of the `while` loop are so common when executing loops a fixed number of times that C++ provides a concise representation – the `for` repetition control statement:

```
for(i=0; i<N; i++)
{
   cin >> number;
   total = total + number;
}
```

This control structure has exactly the same effect as the `while` loop listed above.

The following is a simple example of a `for` loop with an increment statement using the increment operator to calculate the sample mean and variance of $N$ numbers. The `for` loop is executed $N$ times. [6]

```
//ForStatistics.cc
//Computes the sample mean and variance of N numbers input at the keyboard.
//N is specified by the user but must be 10 or fewer in this example.

#include <iostream>
using namespace std;

int main()
{
   int    i, N=0;
   float  number, sum=0.0, sumSquares=0.0;
   float  mean, variance;

   // Wait until the user inputs a number in correct range (1-10)
   // Stay in loop if input is outside range
   while(N<1 || N>10)
   {
      cout << "Number of entries (1-10) = ";
      cin  >>   N;
   }

   // Execute loop N times
   // Start with i=1 and increment on completion of loop.
   // Exit loop when i = N+1;
   for(i=1; i<=N; i++ )
   {
       cout << "Input item number " << i << " = ";
       cin >> number;
       sum = sum + number;
       sumSquares = sumSquares + number*number;
   }

   mean = sum/N;
   variance = sumSquares/N - mean*mean;

   cout << "The mean of the data is " << mean <<
          " and the variance is " << variance << endl;
   return 0;
}
```

[6] *Double click* on the icon with the file name `ForStatistics.cc` and compile and run the program. Change the value of `N` (both negative and positive numbers) to make sure you understand the operation of both `while` and `for` loops.

A more general form of the `for` repetition control statement is given by:

`for(` *statement1*`;` *boolean-expression*`;` *statement3* `)`
{
    *statement2*`;`
}

The first statement (*statement1*) is executed and then the boolean expression (condition) is tested. If it is true *statement2* (which may be more than one statement) and *statement3* are executed. The boolean expression (condition) is then tested again and if true, *statement2* and *statement3* are repeatedly executed until the condition given by the boolean expression becomes false.

# Part IA Computing Course — Session 2

## A. Objectives

After reading through section 6 of the tutorial guide and studying and executing the C++ programs in the examples (boxed) you should now be familar with:

□ Boolean expressions with relational operators
□ Simple control structures for selection and repetition

You will now use these skills in the following computing exercises. The information in section C will help you to think about what is involved in producing a working solution.

## B. Computing Exercises

### Simultaneous equations

**(4)** Write and test a program to solve two linear equations with real coefficients of the form:

$$\begin{aligned} a_1x + b_1y &= c_1 \\ a_2x + b_2y &= c_2 \end{aligned}$$

The coefficients of each equation should be input by the user via the keyboard.

Your program should check that the two equations are independent (i.e. test the condition that the equations have a unique solution) and display the solution.

### Estimating the value of $\pi$

**(5)** Write a program to print out the first $N$ $(1 < N \leq 100)$ terms in the series:

$$\sum_{i=1}^{N} \frac{1}{i^2} = 1 + \frac{1}{4} + \frac{1}{9} + \frac{1}{16} \cdots \frac{1}{N^2}$$

The number of terms, $N$, is to be input by the user at the keyboard.

Modify your program so that for each value of the index $i$, it evaluates the sum of the first $i$ terms.

The sum of this series can be shown (Leonhard Euler (1707-1783)) to converge to $\pi^2/6$. Make another modification to your program so that at each *iteration* the estimate of $\pi$ is printed instead of the sum. How good is the estimate of $\pi$ after `N=100` iterations? How many iterations are needed to get an estimate of $\pi$ which is accurate to 2 decimal places after rounding?

## C. Notes on implementation of exercise (4)

1. **Getting started:**

   Begin by writing a simple program that prompts the user for the real coefficients $(a, b, c)$ of the two linear equations in the form $ax + by = c$.

   You will need to *declare* variables to store the coefficients of each equation (e.g. `a1`, `b1`, ... `c2`).

2. **Testing for a unique solution:**

   The solution of the two simultaneous equations:

$$\begin{aligned} a_1x + b_1y &= c_1 \\ a_2x + b_2y &= c_2 \end{aligned}$$

   is given by (Cramer's Rule):

$$\begin{aligned} x &= \frac{c_1b_2 - b_1c_2}{(a_1b_2 - b_1a_2)} \\ y &= \frac{a_1c_2 - c_1a_2}{(a_1b_2 - b_1a_2)} \end{aligned}$$

   There will not be a unique solution if the denominator is zero, i.e. $(a_1b_2 - b_1a_2) = 0$. Check that the two equations yield a unique solution by evaluating this condition.

3. **Computation of solution using IF/ELSE selection:**

   If the solution is unique then compute and display the solution, else display a message that there is no unique solution.

## D. Notes on implementation of exercise (5)

1. **Getting started:**

   Begin by writing a simple program that prompts the user for the number of terms, `N`, in the range $1 < N \leq 100$. (See the program `ForStatistics.cc` if you are unsure of how to do this). Remember to declare and initialise all the variables that you use and to end your `main()` function with a `return 0` statement. Compile and test your program after each of the following modifications.

2. **Repetition loop:**

   Write a `for` repetition loop to print out the value of the index (counter) `i` and the value of the $i$th term:

   ```
   iterm = 1.0/(i*i);
   ```

   from `i=1` to`i=N`. Note that the index (counter) `i` is an integer. The variable `iterm`, however, must be declared as a `float` and the right-hand side must be evaluated with *floating point* division if the terms are to be evaluated correctly.

3. **Sum of first $i$ terms:**

   Modify the program so that at each iteration (for each value of `i`) it computes and prints the sum of the first $i$ terms.

4. **Estimating the value of $\pi$:**

   An estimate for the value of $\pi$ can be obtained from the sum of $N$ terms, (e.g. `sum`) by computing `sqrt(6.0*sum)`. To use the mathematical library function `sqrt()` you must remember to include the header file `cmath` at the top of your program.

   Print out the estimate of $\pi$ at each iteration. The series converges slowly. How good is the estimate of $\pi$ after `N=100` iterations? Change the maximum value of `N` so that a better accuracy can be achieved (e.g. `N<1000`). The next exercise produces a better estimate for the value of $\pi$ with fewer iterations.

## E. Evaluation and marking

Run your programs and check that they give the correct answers.

Show your working programs for session 1 and 2 and the program source code to a demonstrator for marking. You can print a copy of any file by selecting the `Print` button of the `xcc window`.

Exercises 1–5 **must** be completed and marked **before** your next laboratory session.

# 7 Functions

Computer programs that solve real-world problems are usually much larger than the simple programs discussed so far. To design, implement and maintain larger programs it is necessary to break them down into smaller, more manageable pieces or *modules*. Dividing the problem into parts and building the solution from simpler parts is a key concept in problem solving and programming.

In C++ we can subdivide the functional features of a program into blocks of code known as **functions**. In effect these are subprograms that can be used to avoid the repetition of similar code and allow complicated tasks to be broken down into parts, making the program modular.

Until now you have encountered programs where all the code (statements) has been written inside a single function called `main()`. Every executable C++ program has at least this function. In the next sections we will learn how to write additional functions.

## 7.1 Function definition

Each function has its own name, and when that name is encountered in a program (the **function call**) execution of the program branches to the body of that function. After the last statement of the function has been processed (the **return** statement), execution resumes on the next line after the call to the function.

Functions consist of a header and a body. The **header** includes the name of the function and tells us (and the compiler) what type of data it expects to receive (the *parameters*) and the type of data it will return (*return value type*) to the calling function or program.

The **body** of the function contains the instructions to be executed. If the function returns a value, it will end with a `return` statement. The following is a formal description of the syntax for defining a function:

*return-value-type* *function-name*`(` *parameter-list* `)`
{
    *declaration of local variables*;
    *statements*;

    `return` *return-value*;
}

The syntax is very similar to that of the `main` program, which is also a function. `main()` has no parameters and returns the integer 0 if the program executes correctly. Hence the return value type of the `main()` function is `int`.

```
int main()
{
    declarations;
    statements;

    return 0;
}
```

## 7.2 Example of function definition, declaration and call

Let us first look at an example of program written entirely with the function `main()` and then we will modify it to use an additional **function call**.

We illustrate this with a program to calculate the factorial ($n!$) of an integer number ($n$) using a `for` loop to compute:

$$n! = 1 \cdot 2 \cdot 3 \ldots (n-2) \cdot (n-1) \cdot n$$

```
// MainFactorial.cc
// Program to calculate factorial of a number

#include <iostream>
using namespace std;

int main()
{
   int i, number=0, factorial=1;

   // User input must be an integer number between 1 and 10
   while(number<1 || number>10)
   {
     cout << "Enter integer number (1-10) = ";
     cin >> number;
   }

   // Calculate the factorial with a FOR loop
   for(i=1; i<=number; i++)
   {
      factorial = factorial*i;
   }

   // Output result
   cout << "Factorial = " << factorial << endl;
   return 0;
}
```

Even though the program is very short, the code to calculate the factorial is best placed inside a function since it is likely to be executed many times in the same program or in different programs (e.g. calculating the factorials of many different numbers, computing binomial coefficients and permutations and combinations).

```
// FunctionFactorial.cc
// Program to calculate factorial of a number with function call

#include <iostream>
using namespace std;

// Function declaration (prototype)
int Factorial(int M);

int main()
{
   int number=0, result;

   // User input must be an integer number between 1 and 10
   while(number<1 || number>10)
   {
     cout << "Integer number = ";
     cin >> number;
   }

   // Function call and assignment of return value to result
   result = Factorial(number);

   //Output result
   cout << "Factorial = " << result << endl;
   return 0;
}


// Function definition (header and body)
// An integer, M, is passed from caller function.
int Factorial(int M)
{
   int i, factorial=1;

   // Calculate the factorial with a FOR loop
   for(i=1; i<=M; i++)
   {
      factorial = factorial*i;
   }

   return factorial; // This value is returned to caller
}
```

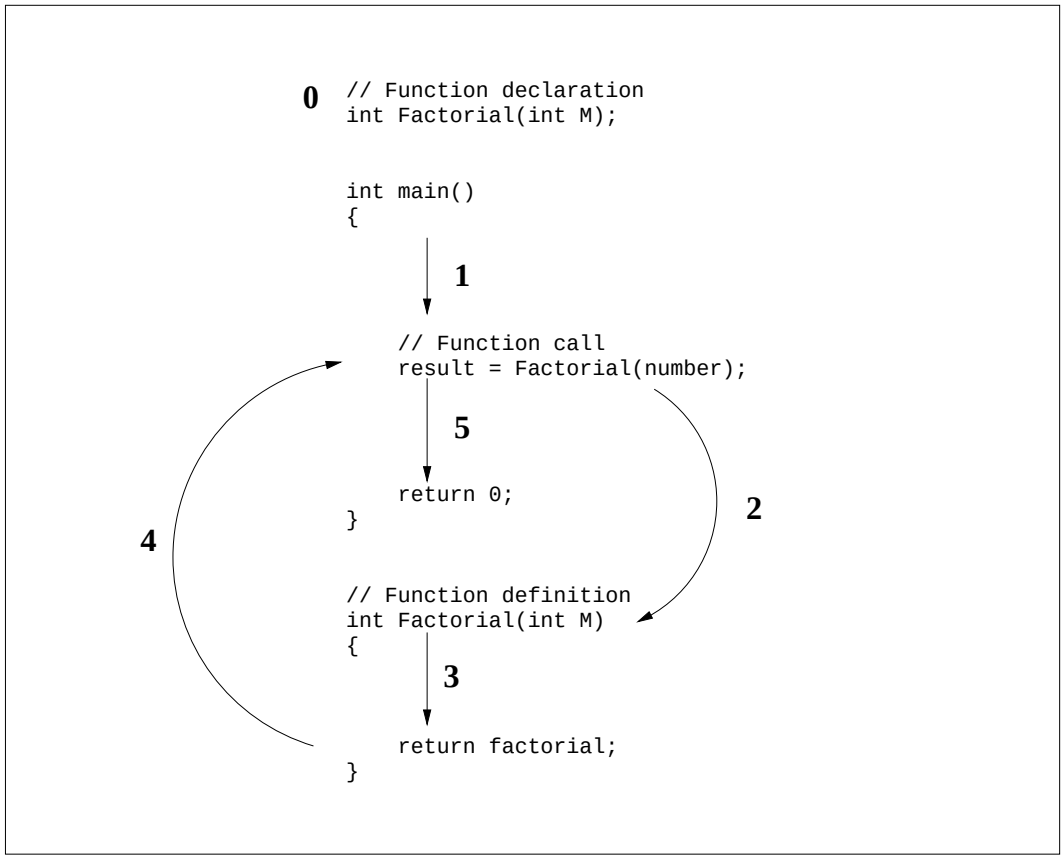
Three modifications to the program have been made to incorporate a function. If we look at the modified sample program, `FunctionFactorial.cc`, we find:

1. The **declaration** of the function above the main program. The declaration (also known as the *prototype*) tells the compiler about the function and the *type* of data it requires and will *return* on completion.

2. The **function call** in the main body of the program determines when to branch to the function and how to return the value of the data computed back to the main program.

3. The **definition** of the function `Factorial()` below the main program. The definition consists of a **header** which specifies how the function will interface with the main program and a **body** which lists the statements to be executed when the function is called.

Before a function can be used it must be declared (0), usually at the top of the program file. When the function name is encountered in the main body of the program (1), execution branches to the body of the function definition (2). Copies of the values of function arguments are stored in the memory locations allocated to the parameters. The statements of the function are then executed (3) up to the first `return` statement when control returns to the main body (4). Any value returned by the function is stored by an assignment statement. Execution in the main body is resumed immediately after the function call (5).



## 7.3 Function header and body

The function is defined below the body of `main()`. The **header** in this example:

```
int Factorial(int M)
```

indicates that the `Factorial()` function expects to be **passed** an integer value (the *parameter type*) from the `main` body of the program and that the value passed will be stored locally in a variable named `M` (the *formal parameter name*). The *return value type* of the function is also `int` in this example, indicating that at the end of executing the body of the function, an integer value will be returned to the statement in which the function was called. Functions which do not return a value have *return value type* `void`.

The **body** of the function computes the factorial of a number in exactly the same way as in the example with only a `main()` function. The execution of the function terminates with a *return statement*:

```
return factorial;
```

which specifies that the value stored in the function variable `factorial` should be passed back to the calling function.

## 7.4 Function declaration

Every function has to be **declared** before it is used. The declaration tells the compiler the name, return value type and parameter types of the function. In this example the declaration:

```
int Factorial(int M);
```

tells the compiler that the program passes the value of an integer to the function and that the *return value* must be assigned to an integer variable. The declaration of a function is called its **prototype**, which means the "first" time the function is identified to your program.

The function prototype and the function definition must agree exactly about the return value type, function name and the parameter types. The function prototype is usually a copy of the function header followed by a semicolon to make it a declaration and placed before the main program in the program file.

## 7.5 Function call and execution

The function definition is entirely passive. By itself it does nothing unless instructed to execute. This is done by a statement in the main program called the **function call**.

For example the statement:

```
result = Factorial(number);
```

calls the function `Factorial()` and passes a copy of the value stored in the variable, `number`. When the function is called, computer memory is allocated for the parameter, `M` and the value passed is copied to this memory location. Memory is also allocated to the (local) variables `factorial` and `i`. The statements of the function are then executed and assign a value to the variable `factorial`. The `return` statement passes this value back to

the calling function. The memory allocated to the parameters and local variables is then destroyed. The value returned is assigned to the variable on the left-hand side, `result`, in the expression used to call the function. The net effect of executing the function in our example is that the variable `result` has been assigned the value of the factorial of `number`.

A function can be called any number of times from the same or different parts of the program. It can be called with different parameter values (though they must be of the correct type). For example the following fragment of code can be used to print out the factorials of the first 10 integers:

```
for(i=1; i<=10; i++)
{
   result = Factorial(i);
   cout << i << "! = " << result << endl;
}
```

and:

```
binomialCoefficient = Factorial(n)/(Factorial(k) * Factorial(n-k));
```

can be used to compute the binomial coefficient:

$$\frac{n!}{k!(n-k)!}$$

## 7.6 Function arguments

The names of variables in the statement calling the function will not in general be the same as the names in the function definition, although they must be of the same type. We often distinguish between the **formal parameters** in the function definition (e.g. `M`) and the **actual parameters** for the values of the variables passed to the function (e.g. `number` in the example above) when it is called.

Function arguments (actual parameters) can include constants and mathematical expressions. For example the following statement assigns the value `24` to the variable `result`.

```
result = Factorial(4);
```

The function arguments can also be functions that return a value, although this makes the code difficult to read and debug.

## 7.7 Another example

The following is another example of the declaration, definition and call of a function, `AreaTriangle()`, in a program to calculate the area of a regular hexagon inscribed in a circle of radius input by the user.

```
// HexagonValue.cc
// Program to calculate the area of a regular hexagon inscribed in a
// circle as sum of areas of 6 triangles by calling AreaTriangle()

#include <iostream>
using namespace std;

// AreaTriangle function prototype
float AreaTriangle(float base, float height);

int main()
{
   float side, segmentHeight, hexagonArea;
   float cosTheta = 0.866025;

   cout << "Program to calculate the area of a hexagon" << endl;
   cout << "Enter side of hexagon: ";
   cin >> side;

   // Base of triangle is equal to side, height is side*cos(30)
   segmentHeight = side*cosTheta;

   // Function returns area of segment. 6 segments for total area.
   hexagonArea =  6.0 * AreaTriangle(side,segmentHeight);

   cout << "Area of hexagon = " << hexagonArea << endl;
   return 0;
}

// AreaTriangle function definition
float AreaTriangle(float base, float height)
{
   float area;

   area = (base*height)/2.0;
   return area;
}
```

The statement:

```
hexagonArea =  6.0 * AreaTriangle(side,segmentHeight);
```

calls the function to calculate the area of a triangle with base and height given by the

values stored in `side` and `segmentHeight` and then assigns the value of 6 times the area (the return value of the function) to the variable `hexagonArea`. It is therefore equivalent to the following:

```
segmentArea = AreaTriangle(side,segmentHeight);
hexagonArea = 6.0*segmentArea;
```

## 7.8 Passing by value or reference

There are two ways to pass values to functions. Up to now we have only looked at examples of **passing by value**. In the *passing by value* way of passing parameters, a copy of the variable is made and passed to the function. Changes to that copy do not affect the original variable's value in the calling function. This prevents the accidental corrupting of variables by functions and so is the preferred method for developing correct and reliable software systems. One disadvantage of *passing by value* however, is that only a single value can be returned to the caller. If the function has to modify the value of an argument or has to return more than one value, then another method is required.

An alternative uses **passing by reference** in which the function is told where in memory the data is stored (i.e. the function is passed the memory address of the variables). In passing the address of the variable we allow the function to not only read the value stored but also to change it. On the other hand, by passing the value by name we simply let the function know what the value is.

To indicate that a function parameter is *passed by reference* the symbol `&` is placed next to the variable name in the parameter list of the function definition and prototype (but **not** the function call). Inside the function the variable is treated like any other variable. Any changes made to it, however, will automatically result in changes in the value of the variable in the calling function.

Passing by reference is an advanced topic in C++. What is really being passed is a *pointer* to the *memory address* of the variable. The symbol `&` is called the **address of** operator and the parameter value becomes the address of the variable following it. The use of *pointers* in C++ is a powerful feature of the language, but is probably one of the most difficult to understand. The preferred way of passing arguments to a function is by *passing by value* calls. This is sufficient for the simple programs considered in this course in which only a single value has to be returned to the calling function. However, if the function has to modify the value of an argument or has to return more than one value, then this must be done by *passing by reference*.

A simple example of passing by reference is given below for a function which swaps the values of two variables in the calling function's data by reading and writing to the memory locations of these variables. Note that the parameters are mentioned only by name in the function call. This appears to be the same as *calling by value*. The function header and prototype, however, must use the `&` symbol by the variable name to indicate that the call is by reference and that the function can change the variable values.

```
// SortReference.cc
// Program to sort two numbers using call by reference.
// Smallest number is output first.

#include <iostream>
using namespace std;

// Function prototype for call by reference
void swap(float &x, float &y);

int main()
{
   float a, b;

   cout << "Enter 2 numbers: " << endl;
   cin >> a >> b;
   if(a>b)
     swap(a,b);

   // Variable a contains value of smallest number
   cout << "Sorted numbers: ";
   cout << a << " " << b << endl;
   return 0;
}

// A function definition for call by reference
// The variables x and y will have their values changed.

void swap(float &x, float &y)
// Swaps x and y data of calling function
{
   float temp;

   temp = x;
   x = y;
   y = temp;
}
```

# 8 Math library and system library built-in functions

Functions come in two varieties. They can be defined by the user or built in as part of the compiler package. As we have seen, user-defined functions have to be declared at the top of the file. Built-in functions, however, are declared in **header files** using the `#include` directive at the top of the program file, e.g. for common mathematical calculations we include the file `cmath` with the `#include <cmath>` directive which contains the *function prototypes* for the mathematical functions in the `cmath library`.

## 8.1 Mathematical functions

Math library functions allow the programmer to perform a number of common mathematical calculations:

| Function | Description |
|---|---|
| sqrt(x) | square root |
| sin(x) | trigonometric sine of x (in radians) |
| cos(x) | trigonometric cosine of x (in radians) |
| tan(x) | trigonometric tangent of x (in radians) |
| exp(x) | exponential function |
| log(x) | natural logarithm of x (base e) |
| log10(x) | logarithm of x to base 10 |
| fabs(x) | absolute value (unsigned) |
| ceil(x) | rounds x up to nearest integer |
| floor(x) | rounds x down to nearest integer |
| pow(x,y) | x raised to power y |

## 8.2 Random numbers

Other **header files** which contain the function prototypes of commonly used functions include `cstdlib` and `time`. These contain functions for generating random numbers and for manipulating time and dates respectively.

The function `random()` *randomly* generates an integer between 0 and the maximum value which can be stored as an integer. Every time the function is called:

```
randomNumber = random();
```

a different number will be assigned to the variable `randomNumber`. Each number is supposed to have an equal chance of being chosen each time the function is called. The details of how the function achieves this will not be discussed here.

Before a random number generator is used for the first time it must be initialised by giving it a number called the *seed*. Each seed will result in a different sequence of numbers. The function `srandom()` is used to provide a seed to initialise the random number generator, `random()`. It must be called with an arbitrary integer parameter (i.e. the seed) which can be conveniently generated by using the value returned by the system clock function `time()` with the actual parameter `NULL`. This returns the calendar time in seconds, converted to an integer value. The following call, which is usually used only once, can be used to initialise the random number generator:

```
srandom(time(NULL));
```

The following program uses these system functions and defines a function to simulate rolling a six-sided die. The die is rolled $N$ times (using a `for` loop) and the proportion of times the value 6 appears as the outcome is calculated. [7].

[7] *Double click* on the icon with the file name `RollDice.cc`. Compile and run the program.

```
// RollDice.cc
// Program to simulate rolling a die with 6 faces N times.
// Output is generated by random number generator and converted to range.
// Fraction of times the number 6 appears is calculated.

#include <iostream>
#include <cstdlib>
#include <ctime>
using namespace std;

int RollDie();

int main()
{
   int i, outcome, N=0, count_six=0, count_other=0;
   float fraction_six, fraction_other;

    // Initialise random number generator with value of system time.
    srandom(time(NULL));

    // Get user input in correct range.
    while(N<1 || N>1000)
    {
       cout << "Input the number of experiments (1-1000): ";
       cin >> N;
    }

    // Perform N experiments.
    // Call RollDie() N times and record number of sixes.
    for(i=0; i< N; i++)
    {
       outcome = RollDie();
       cout << outcome << endl;
       if(outcome==6) count_six++;
       else count_other++;
    }

    //Integer variables must be converted (cast) for correct division
    fraction_six = static_cast<float>(count_six)/N;
    fraction_other = static_cast<float>(count_other)/N;

    // Output results
    cout << "Fraction of outcomes in which 6 was rolled: "
         << fraction_six << endl;
    cout << "Fraction of outcomes in which other numbers were rolled: "
         << fraction_other << endl;
    return 0;
}
```

```
// Function to simulate rolling a single 6-sided die.
// Function takes no arguments but returns an integer.
// Each call will randomly return a different integer between 1 and 6.

int RollDie()
{
   int randomNumber, die;

   randomNumber = random();
   die = 1 + (randomNumber % 6);
   return die;
}
```

# Part IA Computing Course — Session 3

## A. Objectives

After reading through sections 7 and 8 of the tutorial guide and studying and executing the C++ programs in the examples (boxed) you should now be familar with:

☐ Definition, declaration and calling of functions
☐ Passing values to and returning values from functions
☐ Math and System library functions

You will now use these skills in the following computing exercises. The information in sections C and D will help you to think about what is involved in producinga working solution. Suggestions for the *algorithms* (mathematical recipes) and their implementation in C++ are provided.

## B. Computing Exercises

### Finding a solution to $f(x) = 0$ by iteration

**(6)** Write a function that computes the square root of a number in the range $1 < x \leq 100$ with an accuracy of $10^{-4}$ using the Bisection method. The Math library function *must not* be used.

Test your function by calling it from a program that prompts the user for a single number and displays the result.

Modify this program so that it computes the square roots of numbers from 1 to 10. Compare your results with the answers given by the `sqrt()` mathematical library function.

### Finding bounds on the value of $\pi$

**(7a*)** Write a program to compute the bounds on the value of $\pi$ given by the area of regular N-sided polygons inscribing and circumscribing a circle of unit radius (Archimedes' method). Your program must call a function to calculate the sine of a half-angle from the cosine of the angle. *Trigonometric tables and in-built trigonometric functions are not to be used.*

To what accuracy was Archimedes able to estimate $\pi$ when he calculated the area of a 96-sided polygon inscribed in a unit circle and the area of a 96-side polygon circumscribing the unit circle?

**(7b*)** Modify your program so that it aims to compute the value of $\pi$ to a greater precision by making further iterations (e.g. up to $N = 6144$). Do the inner and outer areas converge to a more accurate value of $\pi$?

For $N > 96$ rounding-errors are introduced due to the finite precision in which numbers are stored. Some of these errors can be avoided by changing the variable type from `float` to one with a higher precision, `double`. Make this change to the declaration of all the floating point variables in your program. Do the inner and outer areas now converge to the value of $\pi$? What value of $N$ gives a value of $\pi$ which is accurate to 5 decimal places?

## C. The Bisection Method

The problem of finding the square root of a number, $c$, is a special case of finding the root of a non-linear equation of the form $f(x) = 0$ where $f(x) = c - x^2$. We would like to find values of $x$ such that $f(x) = 0$.

A simple method consists of trying to find values of $x$ where the function changes sign. We would then know that one solution lies somewhere between these values. For example: If $f(a) \times f(b) < 0$ and $a < b$ then the solution $x$ must lie between these values: $a < x < b$. We could then try and narrow the range and hopefully converge on the true value of the root. This is the basis of the so-called Bisection method.

The Bisection method is an iterative scheme (repetition of a simple pattern) in which the interval is halved after each iteration to give the approximate location of the root. After $i$ iterations the root (lets call it $x_i$, i.e. $x$ after $i$ iterations) must lie between

$$a_i < x_i \leq b_i$$

and an approximation for the root is given by:

$$p_i = \frac{(a_i + b_i)}{2}$$

where the error between the approximation and the true root, $\epsilon_i$, is bounded by:

$$\epsilon_i = \pm\frac{(b_i - a_i)}{2} = \pm\frac{(b_1 - a_1)}{2^i}.$$

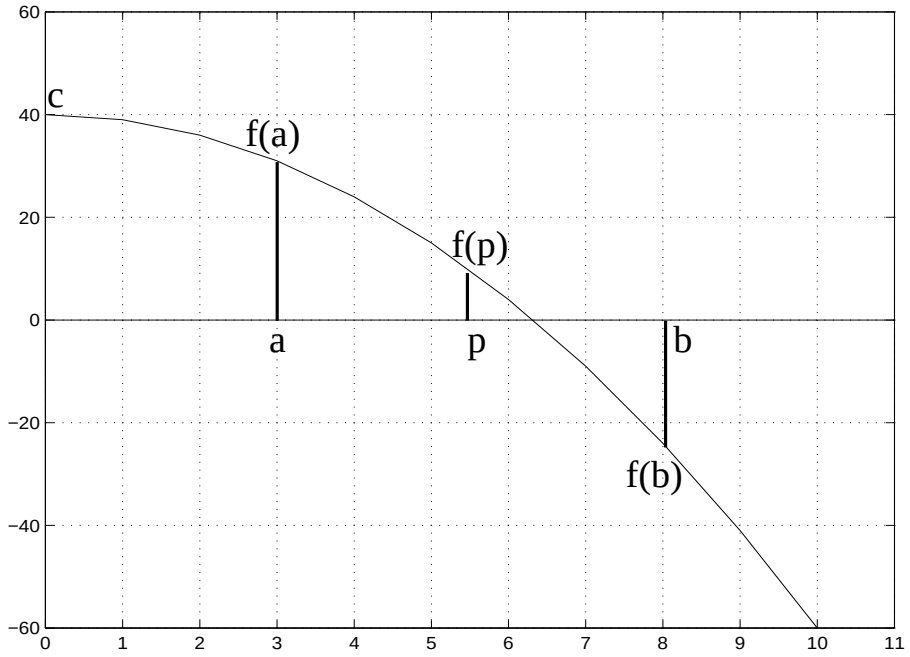
At each iteration the sign of the functions $f(a_i)$ and $f(p_i)$ are tested and **if** $f(a_i) \times f(p_i) < 0$ the root must lie in the half-range $a_i < x < p_i$. Alternatively the root lies in the other half (see figure). We can thus update the new lower and upper bound for the root:

`if` $f(a_i) \times f(p_i) < 0$ then $a_{i+1} = a_i$ and $b_{i+1} = p_i$

`else` $a_{i+1} = p_i$ and $b_{i+1} = b_i$

The Bisection method is guaranteed to converge to the true solution but is slow to converge since it only uses the sign of the function. An alternative is the Newton-Raphson method which takes into account the gradient of the function, $f'(x)$, and only needs one starting guess for the root. In the Newton-Raphson method the next approximation to the root is given by:

$$p_{i+1} = p_i - \frac{f(p_i)}{f'(p_i)}$$




## D. Notes on algorithm and implementation for exercise (6)

The square root of a number will be found by calling a user-defined function to implement one of the iterative algorithms (i.e. repetition of a pattern of actions) described above. Review tutorial section 7 which describes how to define a function and how to pass parameters to the function and return values to the main program.

You are required to define a function to find the square root of a number i.e. $f(x) = c - x^2$. Your solution is to be accurate to 5 decimal places. Since the number input by the user is between 1 and 100 the root will satisfy $0 < x \leq 10$ and a valid initial guess for the lower and upper bound for the solution will always be $a_1 = 0.1$ and $b_1 = 10.1$. The error after $i$ iterations will be $\pm\frac{10}{2^i}$. To produce a solution which is accurate to 5 decimal places we will need more than 20 iterations.

1. **Getting Started:**

   Start with a very simple program which prompts the user for the value of a real number in the range $1 < c \leq 100$.

2. **Function definition**

   You are required to **define** a *function*, `MySquareRoot()`, which is *passed* the number (i.e. single parameter of type `float`) and *returns* the approximate value of its square root (i.e. return value type is `float`).

   The C++ code for the function should be placed below the body of `main()`. Begin the implementation of the function by typing in the *function header* and the opening and closing braces. For example:

   ```
   float MySquareRoot(float square)
   {
   ```

```
   // Body of function definition
}
```

3. **Function body and implementation of algorithm**

   Inside the body of the function (i.e. after the opening brace):

   (a) You will need to declare *local variables* to store the values of $a_i$, $b_i$, $p_i$ and $f(a_i) \times f(p_i)$. For example: `lower`, `upper`, `root` and `sign`. Initialize the values of `lower` and `upper` to 0.1 and 10.1 respectively.

   (b) Set up a loop using the `while` or `for` repetition control statements to repeat the following algorithm (Bisection method) at least 20 times.

   (c) In each execution of the loop:

   - Estimate the value of the root as the average of the lower and upper bounds. Store this value in variable `root`.
   - Evaluate the function at the current value of lower (i.e. $a_i$) and at the current estimate of the root, ($p_i$).
   - Evaluate $f(a_i) \times f(p_i)$ and store this value in variable `sign`.
   - Depending on the value of `sign` update the lower and upper bounds by the bisection method described above.

   (d) The function must end with a `return` statement to pass back the approximate value of the square root.

4. **Function declaration:**

   **Declare** the function by including the function prototype before `main()`. Compile your program to make sure you have not made any typing or syntax errors.

5. **Testing of function:**

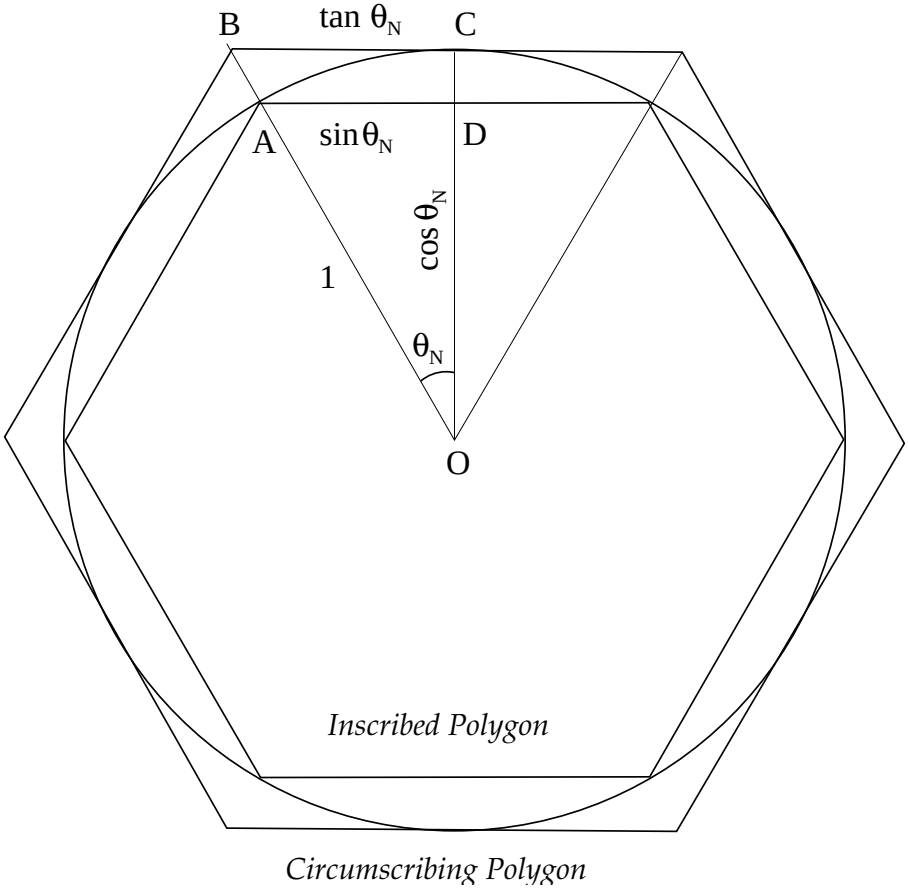
   **Call** your function from your program with the (actual) parameter, e.g. `number`. The *return value* of the function is to be assigned to a variable (e.g. `squareRoot`) which should also be declared in `main()`:

```
squareRoot = MySquareRoot(number);
```

   Test it by calculating the square root of 2 ($\sqrt{2} = 1.41421$) etc. or comparing the result with that given by the library function, `sqrt()`. To use the mathematical library function you must remember to include the header file `cmath` at the top of your program.

6. **Loop**

   Set up a loop in the main routine to *call* the function 10 times to calculate the square roots of the integers $1, 2, \ldots 10$.




## E. Archimedes and the value of $\pi$

The value of $\pi$ was important in the constructions of the ancient world. The Bible (I Kings 7:23) describes Solomon using a value of 3 to calculate the circumference of a circle. This was the average of the area of a square circumscribing the circle and a square inscribed in a circle. In an Egyptian papyrus of 1650BC it was listed as $4(8/9)^2 = 3.16$. The first theoretical calculation was carried out by Archimedes (287-212BC, Sicily), probably the greatest mathematical intellect of the ancient world. Archimedes knew that the value of $\pi$ could not be expressed exactly as a rational number and was able to place a bound on its value by calculating the area of regular polygons inscribing and circumscribing a unit circle.

Archimedes knew that the area of a circle will always be greater than that of an N sided regular polygon inscribed in it. It will also always be less than the area of an N sided regular polygon circumscribing it. For a unit circle the area of an inscribed polygon ($I_N$)with N sides can be calculated as the sum of the area of N triangular segments and is given by:

$$I_N = N\sin(\theta_N)\cos(\theta_N)$$

where $\theta_N$ is half the angle subtended by each isosceles triangle with base given by the side of the polygon (i.e $\theta_N = \pi/N$ radians or $180/N^o$).

The area of the circumscribing polygon ($O_N$) can also be calculated as the sum of the

areas of N triangular segments and is equal to:

$$O_N = N\tan(\theta_N)$$

These two areas give lower and upper bounds on the area of the unit circle and hence on the value of $\pi$:

$$N\sin(\theta_N)\cos(\theta_N) < \pi < N\tan(\theta_N)$$

As N increases, $I_N$ gets larger while $O_N$ gets smaller and their difference diminishes. In the limit as N becomes very large, the difference vanishes and the areas of the polygons give the value of $\pi$. (Note that Archimedes was anticipating Newton and the introduction of differential and integral calculus.)

Archimedes used a similar formula to compute the area of a 96 sided polygon inscribing and circumscribing a circle of unit radius. These two areas gave him a lower and upper bound on the value of $\pi$. Unfortunately he did not have access to the trigonometric tables and functions. He did, however, know the value of the sine, cosine and tangent of $\theta_6 = 30^o$, which he got from looking at an equilateral triangle and using Pythagoras' theorem. He also knew that he could compute the values of sine of $15^o$ by using the half-angle trigonometric relationship:

$$\sin(\theta_N) = \sqrt{\frac{1 - \cos(2\theta_N)}{2}}$$

and the values of cosine and tangent of the angles by using the well-known trigonometric relations.

By using these relations iteratively he was able to compute the trigonometric values of angles down to $(15/8)^o$ and hence calculated the areas of the 96 sided polygons. The computing problem will determine the accuracy of his estimate, which we can assume he took as the average of the upper and lower bound.

## F. Notes on algorithm and implementation for exercise (7*)

We begin by considering the hexagon ($N = 6$ and $\theta_N = 30^o$) and we will write a simple program to calculate the trigonometric values of sine, cosine and tangent of $30^o$ from the cosine of $60^o$. This will be done by calling a user-defined function. Review tutorial section 7 which describes how to define a function and how to pass parameters to the function and return values to the main program.

1. **Getting Started:**

   Start with a very simple program which declares and initialises the variables `cosDoubleTheta = 0.5` and a variable which stores the number of vertices, `N=6`. These must **not** be declared as constants, since their values will be changed later in the full program.

2. **Function definition:**

   **Define** a *function*, `SineHalfAngle()`, which is *passed* the trigonometric value of cosine of an angle (i.e. single parameter of type `float`) and *returns* the trigonometric value of sine of half the angle (i.e. return value type is `float`).

The function must implement the half-angle trigonometric relationship given by:

$$\text{sinAngle} = \sqrt{(1 - \text{cosDoubleAngle})/2}$$

The function will be passed a value of type float which it should assign to `cosDoubleAngle` – the formal parameter. It should then return the value of the variable `sinAngle`.

You should use the mathematical library function `sqrt()` by including the header file `math` before `main()`. Remember to declare any *local* variables (e.g. `sinAngle`) you have used in the implementation of the function.

3. **Function declaration:**

   **Declare** the function by including the function prototype before `main()`. Compile your program to make sure you have not made any typing or syntax errors.

4. **Testing of function:**

   **Call** your function from your program with the (actual) parameter `cosDoubleTheta`. The *return value* of the function is to be assigned to a variable (e.g. `sinTheta`) which should also be declared in `main()`:

   ```
   sinTheta = SineHalfAngle(cosDoubleTheta);
   ```

   Test it by calculating the sine of $30^o$ when it is passed the value of cosine of $60^o$ (i.e. `cosDoubleTheta = 0.5`).

5. **Calculating corresponding cosine and tangent values:**

   Having calculated the sine of the half-angle, `sinTheta`, we can now proceed to calculate the corresponding values of cosine and tangent of the half-angle using the simple trigonometric relations:

   $$\text{cosTheta} = \sqrt{(1 - \text{sinTheta}^2)}$$

   $$\text{tanTheta} = \frac{\text{sinTheta}}{\text{cosTheta}}$$

   You do **not** need to define functions for these. The cosine and tangent values can be evaluated from the simple expressions above.

6. **Bounds on $\pi$:**

   The trigonometic values of sine, cosine and tangent can be used to compute the lower and upper bounds on the value of $\pi$. The lower bound on $\pi$ is given by the area of the inscribed polygon:

   $$N * \text{sinTheta} * \text{cosTheta}$$

   The upper bound on $\pi$ is given by the area of the circumscribing hexagon expressed in terms of the value of the tangent of the angle:

   $$N * \text{tanTheta}$$

Executing your program should now compute the values of the sine, cosine and tangent of $30^o$ which can be used to compute lower and upper bounds on $\pi$. Write a statement to print the value of $N$ and the corresponding values of the lower and upper bounds.

You have now calculated the bounds on $\pi$ for `N=6` by computing the trigonometric values of the sine, cosine and tangent of $30^o$.

7. **Iteration step:**

   To get a better bound on the value of $\pi$ we will double the number of vertices to `N=12` and compute the trigonometric values of sine, cosine and tangent of half of the angle ($15^o$).

   The value of `N` must now be doubled:

   ```
   N = 2*N;
   ```

   and the angle must be halved. The half angle trigonometric values will thus replace the trigonometric values. We do this by assigning the value of `cosTheta` to `cosDoubleTheta`:

   ```
   cosDoubleTheta = cosTheta;
   ```

   Add these statements to your main function.

   To compute the new trigonometric values for N=12 we will need to call our function again to return the value of $\sin 15^o$. This is done by repeating the above statements using a repetition loop.

8. **Repetition loop:**

   We enclose the above sequence of statements in a *repetition loop* to start with the areas (inscribing and circumscribing) of the hexagon (`N=6`, $\theta_N = 30^o$) and to end with the areas of the 96-sided polygons (`N=96`). With the completion of each loop, the value of `N` will be doubled, and the angle will be halved. At the start of the loop the new trigonometric values are calculated. We first calculate the sine value – by calling the function defined – and then the corresponding values of cosine and tangent of the angle.

   You can use either a `for` loop:

   ```
   for(i=0; i<5; i++)
   {
      statement sequence;
   }
   ```

   or a `while` loop:

   ```
   i=0;
   while(i<5)
   {
   ```

*statement sequence*;
`i++;`
}

Make sure that at each iteration you print out the value of `N` and the lower and upper bounds on the value of $\pi$.

After successfully compiling and testing your program make a copy of the source code and show it and the working program to a demonstrator for marking at the end of the next session.

# 9 Arrays

More realistic programs need to store and process a substantial number of items of data. The types of variable considered so far have been simple data types capable of holding a single value.

Arrays are a consecutive group of variables (memory locations) that all have the same type and share a common name. In many applications, arrays are used to represent vectors and matrices.

## 9.1 Declaration

An array is declared by writing the type, followed by the array name and **size** (number of elements in the array) surrounded by square brackets.

| *type* | *array-name*`[` *number-elements* `];` |
|---|---|
| `float` | `position[3];` |
| `int` | `count[100];` |
| `char` | `YourSurname[50];` |

The above examples declare `position` to be an array which has 3 elements which are real numbers (e.g. 3D vector). `YourSurname` contains 50 characters (e.g. storing a surname) and `count` is an array of 100 integer values.

An array can have any legal variable name but it cannot have the same name as an existing variable. The arrays size is fixed when the array is declared and remains the same throughout the execution of the program.

## 9.2 Array elements and indexing

To refer to a particular element of the array we specify the name of the array and the *position number* or **index** of the particular element. Array elements are **counted from 0** and not 1. The first elements will always have position and index 0 and the last element will have index number (position) `N-1` if the array has `N` elements.

The first elements in the arrays declared above are referred to by `position[0]`, `count[0]` and `YourSurname[0]` respectively. The last element of an array declared as `array[N]` with N elements is referred to by `array[N-1]`. The last elements of the example arrays above are therefore referred to by `position[2]`, `count[99]` and `YourSurname[49]` respectively.

The index of an array can also be an expression yielding an integer value.

## 9.3 Assigning values to array elements

Elements of an array can be treated like other variables but are identified by writing the name of the array followed by its index in square brackets. So for instance the statements:

```
marks[1] = 90.0;
scaled[1] = (marks[1] - mean)/deviation;
```

show how the second element of an array called `marks` is assigned a value and how the second element of the array `scaled` is assigned the result of a calculation using this element value.

What is useful about arrays is that they fit well with the use of loops. For example:

```
int count[100];
for(i=0; i< 100; i++)
{
   count[i] = 0;
}
```

can be used to **initialise** the 100 elements of an integer array called `count` by setting them all to 0. The following program uses arrays and repetition loops to calculate the scalar product of two vectors input by the user.

```
// ScalarProduct.cc
// Calculating the scalar product between vectors input by user

#include <iostream>
using namespace std;

int main()
{
   float vectorA[3], vectorB[3], scalar=0.0;
   int i;

   // Get input vectors from user.
   cout << "Enter elements of first vector: " << endl;
   for(i=0;i<3;i++)
   {
      cin >> vectorA[i];
   }
   cout << "Enter elements of second vector: " << endl;
   for(i=0;i<3;i++)
   {
      cin >> vectorB[i];
   }

   // Calculate scalar product.
   for(i=0;i<3;i++)
   {
      scalar = scalar + (vectorA[i] * vectorB[i]);
   }

   // Output result.
   cout << "The scalar product is " << scalar << endl;
   return 0;
}
```

**Note**: care must be taken never to try and assign array elements which are not defined. Its consequences on the execution of the program are unpredictable and hence very difficult

to detect and debug. [8] It is left to the programmer to check that the array is big enough to hold all the values and that undefined elements are not read or assigned.

## 9.4 Passing arrays to functions

### Function definition and prototype

To pass an array argument to a function, the array type and name is included in the formal parameter list (i.e. in the function definition and prototype parameter list). You should also include the square brackets with the size of the array. The latter is optional with one-dimensional arrays, although it might be convenient to pass the number of items in the array as one of the parameters. The following is a function header for a function which receives an array of real numbers.

```
void NormaliseData(float arrayName[], int arraySize)
```

### Function call

In C++ the entire array may be passed using the array name (and **no** square brackets). This looks similar to *pass by value* but is actually very different. The actual parameter passed to the function, the name of the array, is in fact the memory address of the first element of the array. The important point to note here is that (as with passing by reference) when you pass an array to a function you will be able to access and modify the values of any of its elements.

In the following call to the `NormaliseData()` function, a 300 element array, `marks`, is passed to the function:

```
NormaliseData(marks, 300);
```

The function can read and change the value of any of the elements of the array.

Of course if you only need to pass a single element of the array and you want to take advantage of the simplicity and safety of passing by value, this can be done, for example, by having `marks[2]` as an actual parameter in the call to the function. For example in the following function:

```
float ProcessIndividual(float mark);       //Function prototype
scaledMark = ProcessIndividual(marks[2]);  //Function call
```

the call passes the value of `marks[2]` which it stores as a local variable (`mark`). Changes to the value of the local variable will not affect the value in the calling function.

# 10 Vogle Graphics Library

The Vogle (Very Ordinary Graphics Learning Environment) graphics package provides C++ functions for doing simple graphics and plotting data.

[8] When an array of N elements is declared the compiler is told that it expects N elements and sets aside memory for each element. For an array with N elements the array element addressed by `array[N]` is not defined. Unfortunately when `N` is a variable the compiler cannot check this and it cannot therefore prevent the programmer trying to assign this value. In fact it is allowing the programmer to write to or read the memory location of `array[N]` which may contain another variable. The consequences are usually disastrous and the value of a variable in the program will have its value changed without the programmer being aware that this has occurred.

## 10.1 Graphics library functions

The following are some of the Vogle graphics functions that are commonly used. The function prototypes and an explanation of the parameters are listed.

- `void vogleinit(float xlo, float xhi, float ylo, float yhi);`
  Creates a window with default background (WHITE), default foreground (BLACK) and a default margin around the graph. The parameters specify the minimum and maximum x and y values to be plotted.

- `void xaxis(float xlo, float xhi, int nxticks, float y, float ticksize, char title[], float labello, float labelhi);`
  Draw an axis between (`xlo, y`) and (`xhi, y`). The axis should have `nxticks` of length `ticksize`, and a `title` (`char` array) which is placed under the axis. The ticks themselves are labelled with values ranging from `labello` to `labelhi`.

- `void color(int color);`
  Options for `color` include `BLACK`, `RED`, `GREEN`, `YELLOW`, `BLUE`, `MAGENTA`, `CYAN`, `WHITE`.

- `void point2(float x, float y);`
  Draw a point at x, y.

- `void move2(float x, float y);`
  Move graphics position to point (x, y).

- `void draw2(float x, float y);`
  Draw from current graphics position to point (x, y).

- `int getkey();`
  Return the character (ASCII ordinal) of the next key typed at the keyboard.

- `void vexit();`
  Reset the window/terminal (must be the last Vogle function called).

## 10.2 Example to display a one-dimensional array

The following example uses these Vogle graphics library functions to plot the discrete normal distribution. The prototypes of the functions are declared in the following two header files which are included in the usual way at the top of the program file:

```
#include <vogle.h>
#include <vogleextras.h>
```

Discrete samples from the normal distribution are computed from:

$$N(i) = \frac{1}{\sigma\sqrt{2\pi}} \exp - \left[ \frac{(i-\mu)^2}{2\sigma^2} \right]$$

The normal data (61 discrete samples) is stored in a one-dimensional array, `normalData[61]`, which is passed to a user-defined graphics function called `PlotResultsVogle()`, along with the number of elements (`number = 61`) in the function call:

```
PlotResultsVogle(normalData, number);
```

The Vogle library functions `vogleinit(), xaxis(), yaxis(), color(), move2(), draw2()` are used to initialise a graphics window, plot x and y axis, choose a colour for plotting, move to a position on the canvas and then draw to another position in the color defined [9].

```
// PlotNormal.cc
// Program to display a discrete normal distribution using Vogle Graphics

// C++ standard input/output and Math library headers
#include <iostream>
#include <cmath>
using namespace std;

// Vogle Graphics headers for library functions
#include <vogle.h>
#include <vogleextras.h>

// Prototype of user-defined graphics function which receives an array
void PlotResultsVogle(float dataArray[], int number);

int main()
{
   int i, number=61;
   float normalData[61], mean=35.0, deviation;
   float t1, t2;

   // User is prompted for standard deviation. Mean kept constant at 35.
   cout << "Input standard deviation of normal distribution: " << endl;
   cin >> deviation;

   // Calculate discrete normal distribution data.
   // The value of pi is declared in math as M_PI.
   for(i=0; i< number; i++)
   {
     t1 = (i -  mean)/deviation;
     t2 = -0.5*t1*t1;
     normalData[i] = exp(t2)/(deviation*sqrt(2*M_PI));
   }

   // Pass normalData array to Vogle graphical routines.
   PlotResultsVogle(normalData, number);
   return 0;
}
```

[9] *Double click* on the icon with the file name `PlotNormal.cc`. Compile and run the program with different values for the standard deviation ($1 \leq \sigma \leq 15$).

```
// Function to plot the one-dimensional array using Vogle library routines
void PlotResultsVogle(float dataArray[], int number)
{
   // Max and min values of user data to be plotted on graphical window
   float xlo=0.0, xhi = number - 1, ylo = 0.0, yhi=0.25;

   // Initialise Vogle Graphics and scale canvas to user coordinates
   vogleinit(xlo, xhi, ylo, yhi);

   //Draw x and y axes with nxticks and nyticks between max and min values
   int nxticks = 7, nyticks = 6;
   xaxis(xlo, xhi, nxticks, ylo, 0.005, "Outcome", xlo, xhi);
   yaxis(ylo, yhi, nyticks, xlo, 0.005, "Frequency", ylo, yhi);

   // Choose a colour for lines
   color(RED);
   // Plot data as discrete distribution
   for(int i=0; i<number; i++)
     {
       move2(i,0.0);
       draw2(i,dataArray[i]);
     }

   // Wait for key press to indicate that graphics window is to be destroyed
   cout << "Move pointer to vogle window and press any key" << endl;
   getkey();
   vexit();
}
```

# Part IA Computing Course — Session 4

## A. Objectives

After reading sections 9 and 10 of the tutorial guide and studying the example programs you should be able to:

□ Declare and manipulate data with the array data structure
□ Pass arrays to functions
□ Use Vogle graphics library functions to display data

This knowledge will now be used in the following computing exercises which should be completed and marked by the end of the session.

## B. Computing Exercises

### Rolling a six-sided die N times

**(8)** Modify the `RollDice.cc` program to perform $N$ ($N \leq 20000$) experiments and to tabulate the distribution of the outcomes using an array structure.

The value of $N$ is to be specified by input from the keyboard. The output is to be the display of the data as a table of outcome (1-6) and the proportion of experiments in which that outcome value occurred.

### Rolling M dice

**(9)** Define and test a function to simulate a player rolling $M$ dice together ($M \leq 10$). The value of $M$ is to be specified by input from the keyboard.

Modify the program of exercise (8) so that $N$ ($N \leq 20000$) experiments are performed and the distribution of the outcome values (the sum of the $M$ dice values) is displayed graphically.

Comment on the shape of the distributions for $M = 1$, $M = 2$ and as $M$ approaches 10.

### The central limit theorem

**(10*)** For M=10 and N=20000 find the value of the mean and variance of a normal distribution which best approximates the experimental data.

## C. Notes on implementation

The following information offers hints on how to produce a suitable working program. You should start by reviewing the `RollDice.cc` program introduced in section 8 of

the tutorial guide. (Make sure that you understand this program. Ask a demonstrator for help if you are unsure of any of the statements.) A modified copy (which includes the `PlotResultsVogle()` function) called `DisplayRollDice.cc` can be found in the 1AC++Examples folder.

1. **Getting Started:**

   Make a *copy* of the file called `DisplayRollDice.cc` in the 1AC++Examples folder. This program will be modified to allow the number of times an outcome is obtained to be recorded in a one-dimensional array called `count`.

2. **Declaration of array:**

   Although simple variables (e.g. `count_one`, `count_six`) can be used to record the number of times a die value is rolled, an array data structure is more suitable (i.e. `count[1]` or `count[6]` to store the number of times the outcome of an experiment is a one or six).

   The array **index** can be used to specify the die value rolled, (i.e. `outcome`) and should be made to vary from `0` to `6` for convenience. **Declare** a one-dimensional array of size 7 to store the number of occurrences for each `outcome` value.

3. **Initialisation of array:**

   All 7 elements of the array `count[i]` (where $0 \leq i \leq 6$) must be initialised to zero before starting the experiments. This can be done by a repetition loop:

   ```
   for(i=0; i < 7; i++)
   {
      count[i] = 0;
   }
   ```

4. **Recording the outcome of each experiment:**

   The number of occurrences, `count[outcome]`, of a value obtained by rolling the die (`outcome`) can be incremented efficiently by the following statements:

   ```
   outcome = RollDie();
   count[outcome] = count[outcome] + 1;
   ```

   The alternative which uses selection control structures should be avoided since it can not be easily extended when more many dice are rolled together (exercise 9).

   Print out a table of outcome value (1 to 6) and number of occurrences after performing N experiments.

5. **Normalising the counter values:**

   The one-dimensional array of values should be normalised to give a fraction of the total number of experiments. Note that both `count[outcome]` and `N` are integers and the fraction will not be correctly obtained by integer division (see *casting* or example in `RollDice.cc`). You will need to introduce another array of type `float`, e.g. `fraction[i]` such that:

```
fraction[i] = static_cast<float>(count[i])/ N ;
```

The normalised values should be assigned using a repetition loop. Display a table of outcome value and the fraction of outcomes with that value.

## D. Notes on implementation for exercises (9) and (10)

1. **Getting started:**

   The program of exercise (8) will now be modified to include a new function `RollManyDice()` which simulates the rolling of `M` dice together. Computing exercise (9) aims to look at the distribution of `outcome` when `M` dice are rolled together.

   Begin by adding a `while` repetition loop to prompt the user for the value of `M` and check that it is in the correct range (1-10).

2. **Function definition:**

   Define an additional function, `RollManyDice()`, which is *passed* the number of dice rolled, `M`, as a parameter and returns the sum of the dice values as the outcome. The simplest way to do this is to call the `RollDie()` function `M` times using a `for` loop.

   ```
   int i, sumDice = 0;
   for(i=1; i<= M; i++)
   {
      sumDice = sumDice + RollDie();
   }
   ```

   The function definition must end with a return statement. The value returned to the calling function (`sumDice`) will be the outcome of one experiment – the total of the `M` values obtained by the `M` calls to `RollDie()` – and will have an integer value in the range `M` to `6*M`.

3. **Function declaration and call:**

   Declare your function – by adding the function prototype as a statement before the beginning of `main()` – and call it from `main()`. The value returned should be stored in the variable `outcome`. Test your function by displaying the values of the outcomes obtained in the experiments.

   You will have to change the size of the one-dimensional arrays used to store the counters and fractions, `count[i]` and `fraction[i]`, and the repetition loops used to initialise or normalise them. Again for convenience the first M elements will be 0. Note that the size of the array cannot be a variable and you will have to determine the maximum number of elements when rolling M dice. For example if we roll 10 dice the outcomes will have a maximum value of 60. Although the array index will have values between 0 and 60, the array size will be 61.

4. **Graphical display using Vogle library:**

   The data should be displayed using the Vogle library routines. This can be done by passing the `fraction` array to the `PlotResultsVogle()` function defined in `PlotNormal.cc`. If the graphics function definition and prototype are not already in

your program file, use the emacs editor and buffer to copy them into your program file.

You will have to run a large number of experiments (e.g. $N > 10000$) to obtain smooth distributions.

5. **Comparison with the normal distribution**

   Run the `PlotNormal` program in another `xcc` window to plot a discrete normal distribution with mean = 35. Find the value of the standard deviation which best approximates the distribution obtained for M=10 (N=20000).

## E. Marking

Show your working programs for the exercises of session 3 and 4 to a demonstrator for marking **before** the end of your last Michaelmas computing session.

# 11 Advanced Topics

## 11.1 Enumerated constants

There are many examples of data which is not inherently numeric. For example, the days of the week, months of the year, colours. We can refer to such data types by defining symbolic constants and using these symbolic constants in expressions in the program. For example:

```
const int Mon=0, Tue=1, Wed=2, Thu=3, Fri=4, Sat=5, Sun=6;
```

C++ provides a more convenient way for the user to define a new data type. The C++ **enumeration** statement:

```
enum Days {Mon, Tue, Wed, Thu, Fri, Sat, Sun};
```

creates a new variable type with legal values `Mon, Tue, Wed, Thu, Fri, Sat, Sun` which are in fact **symbolic constants** for 0,1,2,3,4,5 and 6. The enumeration is simply assigning an integer to a symbolic constant. The definition makes it convenient to work with days of the week.

Variables can be declared as having the user-defined type and this will ensure that they are only assigned one of the legal values. The following statement declares the variable `day` to have the user-defined type, `Days`.

```
Days   day;
```

## 11.2 Character arrays

A common use of the one-dimensional array structure is to create character strings. A character string is an array of type `char` which is terminated by the `null` terminator character, `'\0'`. The symbol `\0` looks like two characters but represents one. It is called an *escape sequence*.

A character array can be *initialised* on declaration by enclosing the characters in double quotes. The null character will be automatically appended. In the following example the `subject[]` array has the text `Information Engineering` assigned to it, along with the null character. Its size is set to 24 (includes space and null characters) by the compiler. The array `surname` assigned by the `cin >> surname` statement can only be assigned one word of text. The simple input statement stops reading characters when the first *whitespace* character is encountered.

```
// CharacterArray.cc
// Initialisation of a character array and passing to functions

#include <iostream>
using namespace std;

void PrintExtensionNumber(char phoneNumber[]);

int main()
{
   char surname[50];
   char phone[11];
   char subject[] = "Information Engineering";

   // Get user details
   cout <<  "Enter surname and telephone number: " << endl;
   cin  >>  surname >> phone;

   cout <<  surname << " is a student of " << subject << endl;

   // If first two digits are 3 then it is a university number.
   if( phone[0]=='3' && phone[1]=='3')
   {
      cout <<  "Please contact on university extension ";
      PrintExtensionNumber(phone);
   }
   else
   {
      cout <<  "Please contact on external number 9-" << phone[0];
      PrintExtensionNumber(phone);
   }

   return 0;
}

// Function to print out extension number. Ignores first digit.
void PrintExtensionNumber(char phoneNumber[])
{
   int i;

   for(i=1; phoneNumber[i] != '\0'; i++)
   {
     cout << phoneNumber[i];
   }
   cout << endl;
}
```

## 11.3 Multi-dimensional arrays

The arrays that we have looked at so far are all one-dimensional arrays, however, C++ allows multidimensional arrays. For example to declare a two-dimensional array to represent the data of a 3 by 4 matrix called `myMatrix` we could use the following statement and syntax:

```
float  myMatrix[3][4];
```

You may think of the element with indices `i` and `j`, `myMatrix[i][j]`, as the matrix element in the `i` row and `j` column, but remember that array indices always start from `0`.

The following program illustrates the passing of arrays to and from functions. The function `ComputeMatrix()` assigns values to the elements of a 2 by 2 rotation matrix. The actual parameters passed are the name of the rotation matrix (i.e. memory address of the first element) and an expression to determine the rotation angle in radians. The element `matrix[0][1]` is the element of the first row and second column of the matrix.

The function `RotateCoordinates()` computes the coordinates of a point after transformation by the following equation:

$$\begin{bmatrix} x' \\ y' \end{bmatrix} = \begin{bmatrix} \cos\theta & \sin\theta \\ -\sin\theta & \cos\theta \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix}$$

It is passed the name of the matrix and the name of the initial position vector and transformed position vector.

```
// RotationMatrix.cc
// Program to calculate coordinates after rotation

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

void ComputeMatrix(float matrix[2][2], float angle);
void RotateCoordinates(float rot[2][2], float old[2], float transformed[2]);

int main()
{
   float angle, point[2], transformedPoint[2];
   float rotation[2][2];

   // Get angle of rotation and initial position from input.
   cout << "Enter magnitude of rotation about z-axis in degrees: " ;
   cin >> angle;
   cout <<  "Input x and y coordinates: " << endl;
   cin >> point[0] >> point[1];

   // Calculate coefficients of rotation matrix and transform point.
   // The value of pi is declared in math as M_PI.
   ComputeMatrix(rotation, (M_PI*angle/180.0));
   RotateCoordinates(rotation, point, transformedPoint);

   // Output result.
   cout << "The (x,y) coordinates in the rotated system are ";
   cout << transformedPoint[0] << " and " << transformedPoint[1] << endl;
   return 0;
}

void ComputeMatrix(float matrix[2][2], float angle)
{
    matrix[0][0] = cos(angle);
    matrix[0][1] = sin(angle);
    matrix[1][0] = -sin(angle);
    matrix[1][1] = cos(angle);
}

void RotateCoordinates(float rot[2][2], float old[2], float transformed[2])
{
    transformed[0] = (rot[0][0] * old[0]) + (rot[0][1] * old[1]);
    transformed[1] = (rot[1][0] * old[0]) + (rot[1][1] * old[1]);
}
```

## 11.4 Structures

Arrays are examples of data structures in which all the elements must be of the same type. C++ allows the user to define more general data structures, using the keyword `struct` to define a collection of related variables of any type, called a **structure**. For example:

```
struct StudentType{
   char  name[100];
   int   age;
   int   entryYear;
   float marks[5];
   char  college[20];
};
```

defines a new data type called `StudentType`, which is made up of five *fields* or *data members*: two integers, an array of floating point numbers and two character arrays. The body of the structure definition is delineated by braces and must end with a semicolon. The definition is placed at the top of a program file, between include directives and the function prototypes.

Once a structure has been defined, the structure name can be used to declare objects of that type. This is analogous to declaring simple variables.

```
StudentType person;
StudentType firstYear[400];
```

declares the variable `person` and the one-dimensional array, `firstYear[400]` to be of type `StudentType`.

Data members of a structure are accessed and assigned values by specifying the field (data member) name using the *dot operator*, for example:

```
person.age = 19;
firstYear[205].entryYear = 1999;
```

Structures have the advantage that, by collecting related items together, they can be manipulated as single items. For example whole structures can be copied using an assignment statement (unlike arrays):

```
firstYear[24] = person;
```

and manipulated efficiently with repetition control statements:

```
for(i=0; i< 400; i++)
{
   firstYear[i].entryYear = 1999;
}
```

## 11.5 An introduction to object-oriented programming and classes

One of the most important differences between the C++ programming language and other programming languages is the emphasis on the representation of data using programmer or user-defined data types. In C++ an extension to the definition of structures allows the user to include both *data members* (as above) and *member functions* which are allowed to process the data. The encapsulation of data and functions into packages called *objects* of user-defined types called *classes* is a key part of object-oriented programming.

In its simplest form, a `class` is like a structure which includes the definition of functions (member functions or class methods) which are allowed to process the data. See simple example below.

1. Classes and Objects

   The class (e.g. `Date` in the example) can be considered as a specification while the actual item (e.g. `today`) is an instance of a class and is called an *object*. Declaring an object is very similar to declaring a variable:

   *class-name* *object-name*;

2. Accessing data members and functions

   The data members and member functions of the class can be accessed by simply naming them in conjunction with the name of the object they belong to using the *dot operator*.

   *object-name* `.` *item-name*

3. Defining a class member function

   The declaration and definition of class member functions is similar to those used for standard functions: we use function prototypes to declare the functions and place the statements to be executed in a function definition. The only difference is that with a class member function we must tell the compiler which class the function is a member of. This is done by including the name of the class in the function header using a *scope resolution* operator represented by `::`.

   *return-type* *class-name* `::` *function-name*`(` *parameter-list* `)`

4. Restricting access to class members

   One of three levels of access can be specified for each class member (both data and functions) using the keywords `public`, `private` or `protected` followed by a colon. These refer to how the class members may be accessed in a program.

   The `public` members of a class may be accessed directly from anywhere in the program which has access to an object of the class. The `private` members can only be read or modified by class member functions.

### A simple example

We consider a simple example which shows how to create a simple class and define the class's member functions.

```
// SimpleClass.cc
// A program to demonstrate creation and use of a simple class for dates
#include <iostream>
using namespace std;

// Declaration of Date class
class Date {
  public:
  void set(int, int, int);
  void print();

  private:
  int year;
  int month;
  int day;
};

int main()
{
   // Create a Date object called today
   Date today;

   // Call Date member function set()
   today.set(1,9,1999);
   cout << "This program was written on ";
   today.print();
   cout << endl;
   return 0;
}

// Date member function definitions
void Date::set(int d, int m, int y)
{
  if(d>0 && d<31) day = d;
  if(m>0 && m<13) month = m;
  if(y>0) year =y;
}

void Date::print()
{
  cout << day << "-" << month << "-" << year << endl;
}
```

## 11.6 Reading and writing to Files

Storage of data in variables and arrays is temporary. Files are used for permanent retention of large amounts of data. We will now show how C++ programs can process data from files. To perform file processing in C++ the header file `<fstream>` must be included. The latter includes the definitions of `ifstream` and `ofstream` *classes* (special structure definitions). These are used for input from a file and output to a file.

The following command opens the file called, `name.dat`, and reads in data which it stores sequentially in variables, `a`, and `b`:[10]

```
ifstream fin;
fin.open("name.dat");
fin >> a >> b;
```

In a similar way we can write data (in this example, the values of an array) to a file called `output.dat` with:

```
ofstream fout;
fout.open("output.dat");
for(i=0; i<N; i++)
{
   fout << array[i]  << endl;
}
```

Note that `fin` and `fout` are arbitrary names assigned by the programmer [11]. Using `fin` and `fout` highlights the similarity with the simple input and output statements of section 5 which use the input and output stream objects `cin` and `cout` respectively. The syntax of the input and output statements is identical. The only difference is that care must be taken to ensure the files are opened and closed correctly, i.e. that the file exists and is readable or writable. This can be checked by testing the value of `fin.good()` or `fout.good()`. These will have the values `true` if the files are opened correctly.

The following fragment of a program reports an error if the file name specified was not found or could not be opened and prompts the user for another file name.

[10] The *object* called `fin` is declared to be of type (*class*) `ifstream`. The *member function* called `open()` is used to associate the file name, `name.dat`, with the object name, `fin`.

[11] They are known as `ifstream` and `ofstream` *objects* respectively.

```
// OpenFile.cc
// Program to read data from a file. File name is input by user.

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

int main()
{
   char fileName[80];

   // Get filename from keyboard input
   cout << "Enter input file name: ";
   cin >> fileName;

   // Declare fin to be of type (class) ifstream
   // Associate file name with fin
   ifstream fin;
   fin.open(fileName);

   // Prompt for new file name if not able to open
   while(fin.good() == false)
   {
      cout << "Unable to open file. Enter a new name: ";
      cin >> fileName;
      fin.open(fileName);
   }
   return 0;
}
```

After finishing reading from and writing to files they must be **closed** with the statements:

```
fin.close();
fout.close();
```

and the `ifstream` and `ofstream` *objects* `fin` and `fout` can be re-assigned to other file names.

# 12 Further Reading

The tutorial guide has introduced the basic elements of the C++ programming language. The following references provide a comprehensive treatment and useful tips on good programming practice.


1. *C++ How to Program*, Deitel, H.M and Deitel, P.J.
   Prentice Hall, Englewood (NJ), 1996.

2. *Code Complete: A Practical Handbook of Software Construction*, McConnell, S.
   Microsoft Press, Redmond (WA), 1993.