C/C++

A brief and incomplete guide

ANDREW L. HAZEL
Department of Mathematics, University of Manchester
Oxford Road, Manchester, M13 9PL, UK

Contents

1 Introduction

"With great power comes great responsibility"

— Pa Kent to a young superman

C and C++ are perhaps best described as middle-level computer languages. They still allow direct manipulation of bits, bytes and addresses, a feature of assembly languages, but do not sacrifice structure or portability. In contrast to high level languages, such as BASIC, FORTRAN and Pascal, C/C++ is not very fussy about converting between data types and performs relatively few (i. e. hardly any) run-time error checks; although C++ is better than C in this regard. C/C++ demands greater responsibility from the programmer, but can provide more power and flexibility than higher level languages.

1.1 C/C++ is there a difference?

"C makes it easy to shoot yourself in the foot; C++ makes it harder, but when you do, it blows away your whole leg."

— Bjarne Stroustrup (C++ creator)

C++ was originally conceived as a set of extensions to the C language that add support for object-oriented programming. Although not true in the strictest mathematical sense, C is a subset of C++ and hence a C++ compiler should compile both C and C++ code. The syntax and structure of C is common to both languages and by learning C++ you will also learn a lot of C. That said, the styles of programming C and C++ can be quite different and in this course you will be encouraged to "think C++".

1.2 What is object-orientated programming?

"The goal of all inanimate objects is to resist man and ultimately defeat him."

— Russell Baker

A structured language can hide details of the information and instructions needed to perform specific tasks from the rest of the program. For example, using local variables in subroutines is a use of structure. This compartmentalisation of code and data allows easy upgrades to the program and also makes it easier for many programmers to work on a large project. Global variables, common blocks and goto statements are all anathema to structured programs.

Object-oriented programming imposes a higher level of structure than procedural languages. In general, a problem is broken down into subproblems organised in a hierarchical structure. Each subproblem can be translated into self-contained units called objects, which can contain variables and code operating upon those variables. The buzz words that describe three common traits of object-oriented languages are: *encapsulation*, *polymorphism* and *inheritance*.

Encapsulation binds code and data together into an object that cannot be influenced from outside sources except in very tightly controlled ways.

Polymorphism represents the concept of "one interface, multiple methods". The same interface can be used to do different things for different objects: i.e. define + to add numbers, but perform string concatenation on characters and strings, 'a' + 'b' = 'ab'.

Inheritance allows one object to acquire the properties of another. The new object has all the properties of the old and may add more of its own. An example would be to define a generic object "car" that has a steering wheel, four wheels and an engine. The new object "sports car" inherits all these properties and adds a sun roof, go-faster stripes and a huge stereo.

1.3 The simplest C++ program

"Everything should be made as simple as possible, but not one bit simpler."

— Albert Einstein

In C, the basic elements of structure are functions and every C/C++ program must contain at least one function, **main**. You can think of main as equivalent to the PROGRAM ... END PROGRAM construction in FORTRAN. The simplest C/C++ program is shown below:

```
main() {}
```

It is very boring because there are no instructions, but it will compile and it will run. In C/C++, code blocks are marked by braces, sometimes called curly brackets, `{}`; everything between the braces following `main` will be executed while the program is running. The round brackets are used to specify arguments to functions, see §5. It is perfectly possible to write programs using only the main function, but to do so would be a waste as it fails to take advantage of the more powerful structural features of C++. The outline of a more general C++ program is as follows:

#includes // External library functions
using namespace std; // Use standard functions
generic class declarations
derived class declarations
function definitions
`main() {}`

There are 32 keywords defined by the ANSI C standard and these are shown in Table 1. If you are programming in C, these are the only keywords that you have to remember and, in practice, the working vocabulary is about 20 words. The C++ standard defines an additional 32 keywords, shown in Table 2. C++ is a rapidly evolving language and the number of keywords may still change in the future. Case is important in C/C++ and keywords must be specified in lower case: e. g. else is a keyword, but Else, ELSE and ELSe are not.

auto	double	int	struct
break	else	long	switch
case	enum	register	typedef
char	extern	return	union
const	float	short	unsigned
continue	for	signed	void
default	goto	sizeof	volatile
do	if	static	while

Table 1: The 32 C keywords

asm	export	overload	throw
bool	false	private	true
catch	friend	protected	try
class	inline	public	typeid
const_cast	mutable	reinterpret_cast	typename
delete	namespace	static_cast	using
dynamic_cast	new	template	virtual
explicit	operator	this	wchar_t

Table 2: The 32 additional C++ keywords

1.4 Syntax of the source code

"In the beginning was the word. But by the time the second word was added to it, there was trouble. For with it came syntax..."

— John Simon

The key elements of C/C++ syntax are shown below:

- Semicolon used to mark end of statements
- Case is important
- Totally free form, lines and names can be as long as you like!
- Comments take the form `/* C style comment */` or `// C++ style comment`
- Code blocks are surrounded by braces `{}`

The most important point to remember is that all statements must end with a semicolon `;`. Often, forgetting a semicolon can cause a huge number of compilation errors, particularly if the omission is near the start of a program.

1.5 Compiling and running C/C++

"Some compilers allow a check during execution that subscripts do not exceed array dimensions. This is a help, but not sufficient. First, many programmers do not use such compilers because 'They're not efficient.' (Presumably, this means that it is vital to get the wrong answers quickly.)"

— Kernighan & Plauger, The Elements of Programming Style

By convention, C programs are labelled by a .c extension, C++ programs by .cc, .cpp or .C and header files by .h. Large projects may, and probably should, be split into separate files, see Appendix A. The generic compilation commands on UNIX systems are shown below:

C compiler	`cc file1.c`
C++ compiler	`c++ file2.cc`

Both will produce a compiled output file called **a.out** (assembler.out). There are always numerous extra options to compilers, which can vary greatly from system to system. The most useful, and generic, are

`c++ -o output file.cc`	Renames a.out output,
`c++ -Wall file.cc`	All extra error warnings,
`c++ -g file.cc`	Turn on debugging flags,
`c++ -O file.cc`	Optimise the code.

To run a compiled program merely type the name of the program at the command prompt. It is always safest to prepend the program name by ./ to make sure that the UNIX shell searches in your local directory, e. g.

```
c++ file.cc
./a.out
```

2 Expressions

"Every creator painfully experiences the chasm between his inner vision and its ultimate expression."

— Isaac Bashevis Singer

2.1 Data types

"The primary purpose of the Data statement is to give names to constants; instead of referring to pi as 3.141592653589793 at every appearance, the variable Pi can be given that value with a Data statement and used instead of the longer form of the constant. This also simplifies modifying the program, should the value of pi change."

— FORTRAN manual for Xerox computers

There are five basic data types in C (**char**, **int**, **float**, **double** and **void**). C++ adds two more **bool** and **wchar_t**, but for most practical purposes the basic five will be plenty. The void data type is used to declare a function that does not return a value (a subroutine). The other data types correspond to the FORTRAN types CHARACTER, INTEGER, REAL and DOUBLE. The modifiers **signed**, **unsigned**, **long** and **short** may be used to alter the meaning of the base data types, see Table 3.

Type	**Approx size in bits**	**Minimal Range**
(signed) char	8	-127 to 127
unsigned char	8	0 to 255
(signed) int	16	-32,767 to 32,767
unsigned int	16	0 to 65,535
(signed) short int	16	Same as int
unsigned short int	16	Same as unsigned int
(signed) long int	32	-2,147,483,647 to 2,147,483,647
unsigned long int	32	0 to 4,294,967,295
float	32	Six digits of precision
double	64	Ten digits of precision
long double	80	Ten digits of precision

Table 3: Basic data types and their modifiers

2.2 Variables

"The materials of action are variable, but the use we make of them should be constant."

— Epictetus

A *variable* is a named location in memory, used to store something that may be modified by the program. All variables must be declared before they are used. Example declarations are shown below:

```
int i,j,k;
double a,b;
unsigned int profit;
```

Note that commas may be used to separate variables of the same type and that each line ends with a semicolon. In C, variables must be declared at the beginning of a block of code, before any "action statements", but in C++ they can be declared anywhere. The freedom to declare variables anywhere has led to two different schools of thought on where it is "best" to declare variables. Some consider it good style to declare all the variables at the beginning of a code block; that way, one can easily find all variables used in that block. Others would say that by far the best style is to declare variables immediately before they are used; that way, one has the tightest encapsulation and one never declares variables that one doesn't use. Take your pick!

The initial value of a variable may be assigned when the variable is declared. Uninitialised variables can be a source of mysterious errors, so it's a good idea to initialise your variables if practical:

```
int i=0,j=1,k=2;
double a=12.35648;
char ch='a';
```

Note that characters must be enclosed in single quotes `'`.

2.3 Arrays

"We will have rings and things, and fine array, And kiss me, Kate, we will be married o' Sunday."

— William Shakespeare

An *array* is a just like an array in maths: a collection of variables of the same type, referenced by the same name and a number (index). The easiest way to define an array is to enclose the dimension in square brackets `[]` after the array name.

Code	Interpretation
`\b`	Backspace
`\f`	Form feed
`\n`	New line
`\r`	Carriage return
`\t`	Horizontal tab
`\v`	Vertical tab
`\'`	Single quote
`\"`	Double quote
`\0`	Null
`\\`	Backslash
`\a`	Alert (beep)
`\N`	Octal constant (N is a number)
`\xN`	Hexadecimal constant (N is a number)

Table 4: Special character codes in C/C++

```
int x[100]; // 100 integer array
char string[100]; // 100 character array, or string
```

Individual array entries are then accessed by a single number after the variable name, again in square brackets. **Important note:** In C/C++, arrays are indexed from 0, that is an array `double x[3]` will have the components `x[0]`, `x[1]` and `x[2]`.

The most common type of array is a string: an array of characters terminated by a null. A null is just one of the special character codes available in C/C++. These are also used to represent newlines, tabs, etc and are listed in Table 4.

Numerical arrays are initialised by using braces and commas after the array definition. The same syntax may be used to initialise strings, remembering that single characters are enclosed by single quotes and that the final character must be a null `'\0'`. A more convenient alternative is to enclose the entire string in double quotes, in which case the final null is not needed:

```
double x[3] = {1.0, 2.5, 3.7};
char ch = 'a';
char string1[5] = {'J','u','n','k','\0'};
```



```
char string[100] = "This is a test string.";
```

Multidimensional arrays are declared and initialised in an obvious way:

```
double matrix[10][10];
int 3dtensor[3][3][3] = {{1,2,3},{4,5,6},{7,8,9}};
```

The maximum number of dimensions, if any, is determined by the compiler. C/C++ does not have the same number of intrinsic array functions as FORTRAN, but these may be added by hand or in libraries.

2.4 Operators

"Nothing in the universe is contingent, but all things are conditioned to exist and operate in a particular manner by the necessity of the divine nature."

— Baruch (Benedict) Spinoza

C/C++ has a large number of built-in operators, which may be broadly subdivided into four classes *arithmetic*, *relational*, *logical* and *bitwise*. You have already been introduced to the assignment operator, =. The most general form of assignment is

variable = *expression*;

C/C++ allows conversion between data types in assignments. The code shown below will compile without complaint

```
int i=1;
char ch;
double d;

ch = i; // Conversion from int to char
d = ch; // Conversion from char to double
```

The compiler will usually do the right thing, for example converting from double to integer should give the integer part of the result. Again, this flexibility can give rise to funny errors, be sure to check data types in expressions carefully.

Many variables may be assigned the same value by repeated use of the assignment operator. The following fragment assigns the value zero to x, y and z:

```
x = y = z = 0;
```

2.4.1 Arithmetic Operators

Table 5 lists the C arithmetic operators, which are pretty obvious apart from the last three, `%`, `--` and `++`. The modulus operator, `%`, gives the remainder of integer division,

`-`	Subtraction
`+`	Addition
`*`	Multiplication
`/`	Division
`%`	Modulus
`--`	Decrement
`++`	Increment

Table 5: The C/C++ arithmetic operators

it cannot be used on floating point data types. The increment and decrement operators may seem strange at first: `++` adds 1 to its operand and `--` subtracts 1. In other words, `x = x + 1;` is the same as `++x;` and `x = x - 1;` is the same as `x--;`. Increment and decrement operators can precede or follow the operand and there is a difference between the two when used in expressions.

```
x = 10;
y = ++x; // Increments x and then assigns y; i.e. x = y = 11

x = 10;
y = x++; // Assigns y and then increments x; i.e. x = 11, y = 10
```

Arithmetic operators are applied in the following order:

First	`++ --`
	`-` (unary minus)
	`* / %`
Last	`+ -`

Parentheses () may be used to alter the order of evaluation by forcing earlier evaluation of enclosed elements. i. e. `2 + 3*5 = 17`, but `(2+3)*5 = 25`.

2.4.2 Relational and logical operators

Relational and logical operators rely on concepts of false and true, which are represented by integers in C/C++. False is zero and true is any value other than zero. Every expression involving relational and logical operators returns 0 for false and 1 for true. Table 6 shows all the relational and logical operators used in C/C++.

Relational operators	
`>`	Greater than
`<`	Less than
`>=`	Greater than or equal
`<=`	Less than or equal
`==`	Equal
`!=`	Not equal
Logical operators	
`&&`	AND
`||`	OR
`!`	NOT

Table 6: The C/C++ relational and logical operators

The precedence of relational and logical operators is given below

Highest	`!`
	`> >= < <=`
	`== !=`
	`&&`
Lowest	`||`

Examples of relational operations and their values

```
10 > 5 // True (1)
19 < 5 // False (0)
(10 > 5 ) && (10 < 20) // True (1)
10 < 5 && !(10 < 9) || 3 <= 4 // True (1)
```

These operators are usually used to control the flow of a program and will be further explored in section 4.

2.4.3 Shorthand operators

C/C++ uses a convenient (well some would, and do, say obscure) shorthand to replace statements of the form `x = x + 1`,

```
x += 10; /* is the same as */ x = x + 10;
x -= b;  /* is the same as */ x = x - b;
```

This shorthand works for all operators that require two operands and is often used in professional C/C++ programs, so it is well worth taking the time to become familiar with it.

3 The C++ I/O system

"That deaf, dumb and blind kid sure plays a mean pinball."
— The Who

3.1 Simple I/O

No matter how wonderful your program is in isolation, at some point it'll need to interact with the outside world. I/O or input and output is not an intrinsic part of a C/C++ program. This may seem strange, but remember that C was originally written for operating system hacking and some programs don't need I/O, or use specialised I/O that has to be written by manipulating the bits. Standard interfaces are featured in libraries, which may be incorporated into any C/C++ program. The C and C++ I/O libraries are quite different and this course covers only C++ I/O. External library functions for I/O must be included at the start of the program via the `#include <iostream>` command. An example of simple I/O is shown below

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

main()
 {
  int i;

  cout << "This is output.\n"; //print a string

  cout << "Enter a number: ";
  cin >> i;                     //read in a number

  // output the number
  cout << i << " squared is " << i*i << "\n";
 }
```

`cout` is a *stream* the corresponds to the screen and `cin` is a stream that corresponds to the keyboard. The `<<` and `>>` operators are output and input operators and are used to send output to the screen and take input from the keyboard. Several output operators can be strung together in the same command. This program expects integer input and what you type at the keyboard is automatically converted into an integer.

```
%./a.out
This is output
Enter a number 5
5 squared is 25

% ./a.out
This is output
Enter a number 1.45
1 squared is 1

% ./a.out
This is output
Enter a number a
-1073744332 squared is 6290064
```

3.2 File I/O

File I/O is very similar to the I/O system described above and is also based upon streams. An extra header file must be included for file I/O `#include <fstream>`. An input stream uses the `ifstream` class and an output stream uses the `ofstream` class. The following program writes a simple text file

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

main()
 {
  int i;
  ofstream out("test.out"); // Open an output file called test.out
                            // The output stream is called out
  //Error check
  if(!out)
   {
    cout << "Cannot open file test.out\n";
    return 1; // Return to operating system
   }
```



```
  for(i=1;i<=10;i++)
    {
     out <<  i  << " "<< i*i << endl;
    }

   out.close();
   return 0;
 }
```

Opening a file does not have to be done when the stream is declared. The command `out.open("test.out");` can be used anywhere in the body of the function. The command `endl` is an output stream modifier that outputs a newline and flushes the stream, i.e. writes everything to disk.

4 Flow control

"To stop the flow of music would be like the stopping of time itself, incredible and inconceivable."

— Aaron Copland

4.1 Conditional evaluation of code

4.1.1 if

C supports two main conditional constructs: **if** and **switch**. The general form of the **if** command is

`if(`*expression*`)` *statement*;
`else` *statement*;

where a *statement* may be a single command, a block of commands, enclosed in braces `{}` or nothing at all; the `else` command is optional. If *expression* evaluates to true, anything other than 0, the statement following `if` is executed; otherwise the statement following `else` is executed, if it exists. Let's consider a concrete example:

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

main()
 {
  float a;

  cout << "Please enter a number ";
  cin >> a;

  if(a > 0) cout << a << " is positive\n";
  else cout << a << " is not positive\n";
 }
```

The above program takes a user-entered number and determines whether or not it is positive. If-else commands may be nested and this language feature can be used to add a zero test to the above program:

```
#include <iostream>
```



```
using namespace std;

main()
 {
  float a;

  cout << "Please enter a number ";
  cin >> a;

  if(a > 0) cout << a << " is positive\n";
  else
   if(a == 0) cout << a << " is zero\n";
   else cout << a << " is not positive\n";
 }
```

Important note The test for equality is the `==` relational operator. One of the most common programming mistakes is to use the assignment operator `=` instead of the relational operator `==`.

```
if(a=0) cout << a << " is zero\n";
```

The above code will compile, but it doesn't do what you think! The expression `a=0` assigns the value 0 to the variable `a` and returns true if successful. Thus, instead of testing for zero, the statement automatically sets `a` to zero.

4.1.2 The ? command

The `?` operator is a shorthand for simple if-else statements. The syntax is

expression 1 `?` *expression 2* `:` *expression 3*

and if *expression 1* is true, then *expression 2* is evaluated; if *expression 1* is false *expression 3* is evaluated.

```
x = 10;
y = x > 9 ? 100 : 200;
```

In the above example, `y` is assigned the value 100; if `x` had been less than 9, `y` would have the value 200. In if-else terms the same code would be

```
x = 10;
if(x > 9) y = 100;
else y = 200;
```

For clarity, it is generally better to avoid the `?` operator and write out the full if-else construction. Nevertheless, you might encounter it in programs written by others.

4.1.3 switch

The **switch** command is used to construct multiple-branch selections and tests the value of an expression against a list of integer or character constants. Unlike **if**, **switch** can only test for equality, but it can be useful in certain situations, such a menu operations.

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

main()
 {
  char ch;
  double x=5.0, y=10.0;

  cout << " 1. Print value of x\n";
  cout << " 2. Print value of y\n";
  cout << " 3. Print value of xy\n";

  cin >> ch;

  switch(ch)
   {
    case '1':
             cout << x << "\n";
             break;
    case '2':
             cout << y << "\n";
             break;
    case '3':
```



```
             cout << x*y << "\n";
             break;
    default:
             cout << "Unrecognised option\n";
             break;
  } \\End of switch statement
} \\End of main function
```

The `case` keyword is used to demark individual tests and the `break` keyword breaks out of the switch command at the end of each test segment. The optional `default` block corresponds to the final else, it executes if none of the tests are true. If the `break` command is omitted then the next commands in the switch statement will execute. This can be used to produce the same behaviour for many tests. i. e.

```
switch(ch)
 {
  case '1':
  case '2':
  case '3':
            cout << "1, 2 or 3 pressed\n";
            break;
 }
```

4.2 Iterative execution of commands — Loops

4.2.1 for loops

The most versatile loop construct in C/C++ is the **for** loop, which repeats a block of code a number of times until a *predefined* condition is satisfied. The generic form of this construct is

`for(`*initialisation*`;` *condition*`;` *increment*`)` *statement*`;`

where the *initialisation* command is executed at the start of the first loop. The *condition* is tested at the top of each loop and if it is true then the loop commands are executed. The *increment* command is executed at the end of each loop. This command may be used in place of simple incremental DO loops in FORTRAN, e. g.

```
#include <iostream>
```

```
 using namespace std;

 main()
  {
   int i;

   for(i=1; i <= 10; i++) cout << i << i*i << "\n";
  }
```

The above example prints the numbers 1 to 10 and their squares. In this case the increment operator ++ has been used as the *increment* command. The comma operator allows multiple variables to control for loops:

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

 main()
  {
   int i,j;

   for(i=1, j=20; i < j; i++, j-=5)
    {
     cout << i << " " << j << "\n";
    }
  }
```

produces the output

```
%./a.out
1 20
2 15
3 10
4 5
```

The components of a for loop may be any valid C/C++ commands, or they may be absent, allowing powerful generalisations. For example

```
for(x=0; x!=100; ) cin >> x;
```

will run until the user enters 100.


```
for(;;) cout << "An infinite loop\n";
```

A for loop with no components is an infinite loop, it can only be terminated by using a `break` command somewhere in the loop. Time delay loops may be constructed by omitting the body of a for loop e. g.

```
for(t=0; t < 100; t++); // wait for a while
```

4.2.2 while and do-while loops

Another type of loop is the **while** loop, which has the functional form

`while(`*condition*`)` *statement*`;`

and continues to iterate until the *condition* becomes true. The **while** loop tests the *condition* at the top of the loop and so the *statement* will not execute if the *condition* is initially false. A common use of while loops is for convergence tests

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

 main()
  {
   double num=50;

   while(num > 0.0001)
    {
     num /= 2.0;
    }

   cout << num << "\n";
  }
```

This loop repeatedly halves the variable `num` until it is less than a specified tolerance.

A **do-while** loop is similar to a while loop, but performs the test at the bottom of the loop rather than the top. It has the form

`do{`*statement*`;} while(`*condition*`);`

The `do` loop above could equally well have been a do-while loop:

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

main()
 {
  double num=50;

  do
   {
    num /= 2.0;
   }
  while(num > 0.0001);

  cout << num << "\n";
 }
```

Remember, a do-while loop always forces at least one execution of the commands in the body of the loop, whereas a while loop does not.

4.3 Jump commands

The most common jump command is **return** which is used to return from functions to the main program and is therefore covered in the next section, which discusses functions. The **break** command has already been mentioned when describing the **switch** construct. It has a more general scope, however, and can be used to force the immediate termination of *any* loop. For example

```
for(i=0; i <= 10; i++)
 {
  cout << i;
  if(i==5) break;
 }
```

will only display the numbers 1 to 5 on the screen because `break` overrides the conditional test `i <= 10` in the for loop.

The "opposite" of the **break** command is **continue**, which forces another iteration of the loop, skipping any intermediate code. In a for loop, the increment command is performed and then the conditional test. In while and do-while loops control passes

directly to the conditional tests. This can be used to force the earlier evaluation of the condition.

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

main()
 {
  int flag=0;
  char ch;

  while(!flag)
   {
    cin >> ch; //read in a character

    if(ch=='x')
     {
      flag = 1;
      continue;
     }

    cout << "You have entered the character " << ch << "\n";
  }
 }
```

The preceding program takes input from the keyboard and prints the individual characters entered. If the user enters an x the loop terminates without printing that character. This program also illustrates another common C/C++ programming practice, using integer variables as test variables. If flag is 0, then not flag (`!flag`) is 1 which is true and the while loop executes. If flag is set to 1, not flag will be zero, which is false and the loop terminates. If you understand this program, then you should have no problem with relational expressions in C/C++.

5 Functions

"There is no passion like that of a functionary for his function."

— Georges Clemenceau

Functions are the building blocks of C programs and should be where all the action takes place. In general, a function is passed a number of variables (arguments), operates upon them and then returns a value to the main program. You should already be familiar with the most important function `main()`. The general form of a function definition is shown below

data type `function_name(` *arguments* `)`
`{` *body of function* `}`

The *arguments* to a function take the form of a comma-separated list of type definitions i.e. `(double a, int b, ...)`. A function that does not return a value (a subroutine) should be of the type void.

Let us define a function `square()` that returns the square of the argument:

```
double square(double a)
 {
  return(a*a);
 }
```

This particular function expects a double variable as its argument and returns a double variable. The `return` keyword causes the function to return to the point in the program where it was called and sets the return value, if there is one. The following example shows how to use functions in a C/C++ program.

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

//Declare a function
int product(int a, int b)
 {
  return(a*b);
 }

//main body of program
main()
```



```
 {
  int i,j;

  for(i=1, j=10; i<=10; i++, j--)
   {
   cout << i << " " << j << " " << product(i,j) <<  "\n";
   }
 }
```

The C/C++ specification permits recursive functions, that is functions that call themselves. A good example would be the factorial function

```
int fact(int n)
 {
  int result;

  if(n==1) return(1);
  result = fact(n-1)*n; // Recursive call
  return(result);
 }
```

It is very easy to get into infinite loops with recursive functions, so be careful. Also, recursive functions are rarely more efficient than non-recursive equivalents. That said, the use of recursive functions can be an aid in coding tricky algorithms.

5.1 Mathematical functions

Unlike FORTRAN, C/C++ does not have built-in mathematical functions; instead, they are defined in the math library. In order to use these functions, the header file cmath must be included in the program, just like iostream for I/O. The standard trigonometric, hyperbolic and exponential functions are all present: i. e. `sin()`, `exp()`, `acos()`, `log()`, `tanh()`. Also of use are the function `pow(x,y)` which raises the number `x` to the power `y` and `sqrt()`, the square-root function.

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

```
//Define our own cube root function
double cbrt(double arg)
 {
 double result;

 result = pow(arg,(1.0/3.0));
 return(result);
 }

main()
 {
 int i;
 double x;

 // Loop over the numbers 1 to 10
 for(i=1;i<=10;i++)
  {
  x = i*i*i; // This will transform the integer result to a double
  cout << x << " has the cube root " << cbrt(x) << "\n";
  }
 }
```

In many cases, you must **link** the maths library by appending -lm to the compilation command:

```
c++ file.cc -lm
```

This example also illustrates the use of local variables in functions. The variable `result` is only defined within the function `cbrt()`, it cannot be accessed from `main()` and is a form of *encapsulation*.

5.2 String functions

The library cstring includes a number of useful string manipulation functions. Examples are shown below

strcpy(s1,s2)	Copy s2 into s1
strcat(s1,s2)	Concatenate s2 onto the end of s1
strlen(s1)	Returns the length of s1
strcmp(s1,s2)	Returns 0 if s1 and s2 are the same; returns < 0 if s1<s2; returns > 0 if s1 $>$ s2

5.3 Function prototyping

Function prototyping consists of defining the name and arguments of a function, but without specifying the body of the function. It is just like a normal function definition, but with a semicolon after the *arguments*:

data type `function_name(` *arguments* `);`

Function prototypes, but not necessarily the function itself, must be defined before the function can be called. This formalism permits the definition of functions that call each other, for example:

```
double func1(double a); //Must prototype the function

double func2(double b)
 {
 double tmp;
 tmp = b*func1(b);
 return(tmp);
 }

double func1(double a) //Actual definition of func1
 {
 return(2.0*a);
 }
```

In large projects with multiple source files, header files are used to define function prototypes. These header files are included in other parts of the source code, so that functions defined in one file may be called from another, see Appendix A for more details.

5.4 Arguments to main

The function `main` can take arguments from the command line when the program is called. The standard names for these arguments are `int argc`, the number of command line arguments (including the program name) and `char *argv[]`, which is an array of strings, the arguments themselves. The following program opens a file with name specified by the first argument and writes all the other arguments to the file.

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

main(int argc, char *argv[])
 {
 int i;
 ofstream out;

 //Complain if no filename
 if(argc < 2)
  {
   cout <<  "No filename specified\n";
   return 1;
  }

 //Open the file
 //Note that the first argument is argv[1] (program name is argv[0])
  out.open(argv[1]);

 //Error checking
 if(!out)
  {
  cout << "Cannot open file " << argv[1] <<  "\n";
  return 1;
 }

//Now output other arguments into file
for(i=2;i<argc;i++) out << argv[i];
```



```
//Close file
out.close();
return 0;
}
```

If the above program is run with the following arguments

```
% ./a.out test.out This is a test
```

then a file called *test.out* will be created containing the text `Thisisatest`.

5.4.1 Return value of main

You may have noticed the use of the `return` keyword in the function `main()` and might be wondering exactly what it does. The return value of the main function is the value passed to the operating system on termination of the program. It is an integer and a value of 0 usually corresponds to a successfully completed program. A positive return value corresponds to an error and the higher the return value, the more serious the error. Technically speaking the main prototype should be

```
int main(int argc, char *argv[])
```

but the `int` return type is generally understood in C++. In C, however, the `int` should be included, otherwise you will generate compiler warnings.

6 Pointers

"It is not really difficult to construct a series of inferences, each dependent upon its predecessor and each simple in itself. If, after doing so, one simply knocks out all the central inferences and presents one's audience with the starting-point and the conclusion, one may produce a startling, though perhaps a meretricious, effect."

— Sir Arthur Conan Doyle

Pointers are guaranteed to send the novice C/C++ programmer into wide-eyed shock and trembling fear. They will seem strange at first, but represent a very powerful mechanism for handling data. In short, a pointer is nothing more then a variable that holds a memory address, rather than a value. A pointer points to a data value, rather than being a value itself. Think of memory as a sequence of labelled boxes each containing a piece of paper with a number written on it. You can either be told the number directly (a variable) or find it by knowing which box its in (a pointer).

Many people unfamiliar with pointers often ask why. " Why do I need to worry about pointers in C when FORTRAN does that for me?" The answer lies in the philosophical differences between the two languages. In fact, FORTRAN does use pointers, but hides it from the programmer. C gives the programmer the choice of when to use pointers, or not, with the attendant flexibility (and complexity). In practical terms, using pointers generally increases efficiency and permits the easier management of dynamic data structures. Furthermore, pointers must be used if one wants to alter the arguments passed to a function (call by reference), §6.2.

In C/C++ a pointer is defined by placing a `*` in front of the variable name e. g. `double *p` defines a pointer, `p`, to a double type variable. The following example defines a variable, `a`, and two pointers, `p1`, `p2`. The memory address of `a` is assigned to the pointer `p1` and copied to `p2`.

```
main()
 {
  double a, *p1, *p2;

  a = 10.0;
  p1 = &a;  // The &a means 'the address of a'
  p2 = p1;  // Copy address of a to p2

  cout << a << " has the address " << p1  << "\n";
  cout << "We can use p1 " << *p1 << " to recover the value of a "
```



```
       << a << "\n";

  a++; // Increment a

  cout << "We can also use p2 " << *p2
       << " to recover the value of a " << a << "\n";
 }
```

There are two operators associated with pointers: `&` and `*`, which are, loosely speaking, inverses. The address of the variable `a` is obtained by using the command `&a`, whereas `*p` gives the value of the variable at the address `p`, e.g. `*p1`, `*p2`, as shown in the program above.

6.1 Pointers and arrays

There is a very close relationship between pointers and arrays. In fact, the array name on its own is a pointer to the first element of the array (the address at which the array starts). Arrays are arranged sequentially in memory and so you can use pointer arithmetic to access array elements: `p[1]` is the same as `*p++`. Consider the following example code

```
char str[80], *p;

p = str; // Assign the address of the array to p1

//Now we can use p or str to access the array
cout << p[4] << " is the same as  " << str[4]  << "\n";
cout << " We can also express this by  " << *(p+4)  << "\n";
```

6.2 Pointers and functions

Pointers come into their own when used in conjuction with functions. A function cannot permanently alter the value of an argument unless it is passed as a pointer (call by reference). If the argument is not passed as a pointer, a local copy of the variable is made and destroyed on return from the function (call by value).

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

```
// Function without pointer
void square(double a)
 {
  double result;
  result = a*a;
  a = result;
  cout << " In the function square  " << a  << "\n";
 }

// Function with pointer
void square_pointer(double *a)
 {
  double result;
  result = (*a)*(*a);
  *a = result;
  cout << " In the function square_pointer " << *a  << "\n";
 }

main()
 {
  double a = 5;

  //Call function without pointer
  square(a);
  cout << " After the function square " << a  << "\n";

  //Call function with pointer
  square_pointer(&a); // Note that you need to pass the address of a
  cout << " After the function square_pointer " << a  << "\n";

 }
```

This program produces the output

```
%./a.out
In the function square  25
```



```
After the function square 5
In the function square_pointer 25
After the function square_pointer 25
```

The variable `a` in main is not altered by the function `square()`, which does not work with pointers. In contrast, `a` is permanently altered by `square_pointer()`.

In C/C++, arrays are always passed to functions as pointers. Remember that the pointer to an array is merely the name of the array with no square brackets. The pointers may then be used from within the function to access the array elements directly. An array argument to a function may be specified as usual `int array[10]`, as an array of unspecified length `int array[]` or as a pointer `int *array`. **Note:** C/C++ does not do any bound checking so `int array[10]` is not as safe as it looks!

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

//Dot product of two vectors
double dot_product(int n, double *x1, double x2[])
 {
  int i;
  double result=0.0;

  //Loop over the entries of the vectors
  for(i=0;i<=(n-1);i++)
   {
   result += x1[i]*x2[i];
   }

  return(result);
 }

main()
 {
  double x[3] = {1.0,1.5,3.0}, y[3] = {2.0,5.0,10.0};
  double product;

  //Call the function
```

```
  product = dot_product(3,x,y);
  cout << product;
 }
```

IMPORTANT NOTE: When a multi-dimensional array is used as an argument to a function, only a pointer to the first element is passed. In order to find the elements in memory, the compiler needs to know the dimensions of all indices other than the first. The following is illegal

```
  void test_function(double x[][][]);

  main()
   {
    double x[4][5][6];

    test_function(x);
   }
```

and will generate a compiler error. The legal version would be

```
  void test_function(double x[][5][6]);
```

and is rather limiting. Fortunately, using the dynamic memory allocation capabilities in C/C++ one is able to circumvent this restriction, although it's not entirely trivial and is beyond the scope of this course.

6.3 Dynamic memory allocation

Pointers are used in dynamic allocation of memory. A pointer to the first entry in an array must be defined before memory can be allocated. C++ makes dynamic allocation very easy with the **new** and **delete** operators.

```
main()
 {
  int i;
  double *array; // define a pointer to a double
```



```
  array = new double (100.0); // allocate a single double with
                              // initial value of 100.0

  delete array; // free the memory allocated to array


  array = new double [10]; // allocate an array of 10 doubles
                           // cannot specify initial values for arrays

  //Assign values to the array
  for(i=0;i<10;i++) {array[i] = 10.0*(i+1);}

  delete [] array; // Must use delete [] to free arrays
 }
```

6.4 Warning!

Pointers are a blessing and a curse. They are absolutely necessary for many programs, but an error with pointers can be very hard to track down. The problem is that if you use a bad pointer, you are reading and writing to an unknown piece of memory. In the worst cases, you can overwrite other portions of your code, or even the operating system. In the best case, you won't notice anything wrong. A common symptom of pointer problems is when your code runs fine on one type of machine (e. g. Linux PC), but crashes with a segmentation fault on a different type of machine (e. g. Sun server). **Always** make sure that your pointer points to something before you try and use it. It is often a good idea to null-out pointers (initialise pointers to zero) i.e `double *p=NULL;` in order to minimise the instances of bad pointers.

7 References

"If I had no duties, and no reference to futurity, I would spend my life in driving briskly in a post-chaise with a pretty woman."

— Samuel Johnson

References are a part of C++ only and are related to pointers. A *reference* is an implicit pointer that acts as another name, or alias, for an object. However, once set, a reference cannot be changed to refer to a different object. The most useful application of references is to allow call-by-reference parameter passing without the explicit use of pointers. References are defined by using an ampersand `&` in front of the variable name. Consider the example from §6.2, rewritten to use references:

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

// Function without pointer or reference
void square(double a)
 {
  double result;
  result = a*a;
  a = result;
  cout << " In the function square  " << a  << "\n";
 }

// Function with reference
// The body of the function is exactly the same as square()
void square_ref(double &a) //cf square_pointer(double *a)
 {
  double result;
  result = a*a;
  a = result;
  cout << " In the function square_ref " << a  << "\n";
 }

main()
 {
  double a = 5;
```



```
  //Call function by value
  square(a);
  cout << " After the function square " << a  << "\n";

  //Call function by reference
  square_ref(a); // Note that you just pass the variable
  cout << " After the function square_ref " << a  << "\n";
 }
```

This program produces the output

```
%./a.out
In the function square  25
After the function square 5
In the function square_ref 25
After the function square_ref 25
```

The variable `a` in main is not altered by the function `square()`, which is call by value. In contrast, `a` is permanently altered by `square_ref()`, which is call by reference. Using references is often easier than messing around with pointers, but it is important to remember that pointers underly the reference system.

8 Classes

"The studious class are their own victims: they are thin and pale, their feet are cold, their heads are hot, the night is without sleep, the day a fear of interruption,-pallor, squalor, hunger, and egotism."

— Ralph Waldo Emerson

The C++ **class** is used to define objects and forms the basis of C++ programs. A class must be declared before it can be used:

`class class_name {`
private data and functions
access-specifier :
data and functions
access-specifier :
data and functions
.
.
.
access-specifier :
data and functions
`}` *object list* `;`

The *object list* is optional, but if present it defines objects of the class. Once the class is declared, objects may also be defined anywhere in the code. The *access-specifier* is one of the three keywords **public**, **private** or **protected**. The default access type of classes is **private**, so the data and functions can only be used by objects of the class. **public** items, on the other hand, are accessible by other parts of the program. Finally, the **protected** keyword is essentially the same as **private**, but protected members may be used by any derived classes, §10. Access specification can be changed many times in a class declaration, but, in practice, all private functions are declared at the top and public at the bottom.

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

//Create a class student
class student {
```



```
  char name[80];
  double exam_result;
 public:
   void putname(char *n);
   char *getname();
   void putresult(double mark);
   double getresult();
 };

//Define functions for the class
void student::putname(char *n)
 {
  strcpy(name,n);
 }

char *student::getname()
 {
  return name;
 }

void student::putresult(double mark)
 {
  exam_result = mark;
 }

double student::getresult()
 {
  return exam_result;
 }

main()
 {
  student newton;

  newton.putname("Isaac Newton");
```

```
  newton.putresult(45.2);

  cout <<  "Exam results: " << "\n ";
  cout << newton.getname() << " : " << newton.getresult() << "\n ";
 }
```

The above example has a lot of new features so we'll go over it quite carefully. The first part of the program contains the declaration of the class `student`, which has two data variable: a name (string) and an exam result (double). These are private and only accessible through functions of the class, of which there are four, all public. The class functions are defined exactly as normal C functions, but the class name must be prepended to the function name with the scope resolution operator `::`. That is, the function `getname()` in class `student` is defined in the main program by the name `student::getname()`.

In the main function, one object of class `student` is defined, `newton`. The data of this object are accessed by using the public functions, called by prepending the object name and the dot operator `.` to the function name. To call `putresult` for the object `newton` the command is `newton.putresult(45.2)`. **Remember** that you must call public functions in classes by using a particular *object* name and not the *class* name. In the above example, `student.getname()` will not compile.

If you have public variables in a class, they may also be accessed using the dot operator, of course, this violates the principle of encapsulation!

```
class test {
 public:
   int i,j,k; //Entire program can access these data
 };

main()
 {
  test x, y;

  x.i = 1; x.j = 2; x.k = 3; //Direct access to class data
  y.k = 10; //Not the same as x.k

  cout << x.k << " " << y.k;
}
```


In C++, the keyword `struct` may also be used to define classes. The difference between `class` and `struct` is that, by default, all members of a structure are public, rather than private. In all other respects structures and classes are equivalent. N.B. In C, structures are merely groups of variables, rather than variables and functions.

8.1 Constructors and Destructors

C++ provides a mechanism for the automatic initialisation of objects when they are created. This is achieved through *constructor* functions, which have the same name as the class. *Destructor* functions are called when an object is destroyed and may be used to clean up memory or close files used by that object. The destructor has the same name as the constructor, prepended by a tilde ~.

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

class record {
  char name[100];
  double salary;
 public:
  record(); // constructor
  ~record(); //destructor
   void put_name(char *n);
 };

//Define the constructor function for record class
record::record()
 {
  strcpy(name,"Name not assigned");
  salary = 0.0;
  cout << "New record created\n";
 }

//Define the destructor function for record class
record::~record()
 {
```

```
  cout << name << " removed from records\n";
 }

//Normal class function
void record::put_name(char *n)
 {
  strcpy(name,n);
 }

main()
 {
  record boss; //Constructor called here
  boss.put_name("Prof F. Cat");
  return(0); //Destructor called here
 }
```

The above example create a simple class `record` with a constructor and a destructor. Constructors and deconstructors are generally called from outside the object and so **must** be public; a compilation error will result otherwise. It is also possible to pass parameters to constructors to aid the initialisation of an object. These are defined just as in normal functions:

```
//Define the constructor function for record class
record::record(char *n)
 {
  strcpy(name *n);
  salary = 0.0;
  cout << "New record created\n";
 }

main()
 {
 record boss("Prof F. Cat"), slave = "Mr. B. Licker";
 }
```

The second form of initialisation, normal assignment using `=`, works only if the constructor takes a single argument.

8.2 Copy constructors

If an object is passed to a function then a local copy of the object is created (call by value). In this case, C++ does **not** call the class constructor. If the constructor were called then the local object would be initialised, potentially changing its properties. Instead, C++ allows the specification of a *copy constructor*, called whenever an exact copy of an object is created. The syntax is

```
class_name (const class_name &o){}
```

where `o` is the object being copied. Note that, for efficiency, we are actually passing a reference to the object being copied. The `const` keyword in the function argument means that the argument cannot be changed by the function i. e. the copy constructor cannot reset the reference. A copy constructor for the record class would be

```
//Define the copy constructor function for record class
record::record(const record &old_record)
 {
  strcpy(name,old_record.name);
  salary = old_record.salary;
  cout << "Record copied\n";
 }
```

This function must also be prototyped in the class definition, of course.

The copy constructor is called every time an exact copy of an object is required. For example the command `record new_boss=boss;` will call the copy constructor rather than the constructor. The alternative code `record new_boss; new_boss = boss;` does not use the copy constructor, instead the `=` operator is used. If a copy constructor is not defined the default action is to create a bit-wise copy of the object. This is also the default action of the `=` operator, which can be overloaded if desired, §9.

8.3 Pointers to classes

All the pointer concepts described in §6 carry though to classes, including dynamic allocation. Pointers to classes may be defined in exactly the same manner as any other pointer: i. e., `record *slave_pointer;` creates a pointer to the record class. The members of the class can be accessed using the dot operator, but the syntax is clumsy `(*slave_pointer).name`. An alternative and equivalent syntax is often used to access members from class pointers: `slave_pointer->name`. It is invariably more efficient to pass classes to functions using pointers and so the -> operator is a common feature of many C++ programs.

8.4 Friends

In some cases, it is convenient to allow functions that are not members of a class to have access to private data. These are defined via the keyword **friend** and must be prototyped in the class definition. The following code fragment will not compile if the friend prototype line is omitted.

```
class record {
  char name[100];
  double salary;
 public:
  record(); // constructor
  ~record(); //destructor
   void put_name(char *n);
   friend double average_salary(record *r1, record *r2);
 };

 double average_salary(record *r1, record *r2)
  {
   return((r1->salary + r2->salary)/2.0);
  }
```

If a function were a friend to just one class, then it might as well be a member of the class. In some cases, however, members of a number of different classes might need to be accessed by the same function. The function can be made a friend to all the relevant classes and can then access all the data it needs.

9 Overloading

"The word abandons its meaning like an overload which is too heavy and prevents dreaming. Then words take on other meanings as if they had the right to be young. And the words wander away, looking in the nooks and crannies of vocabulary for new company, bad company."

— Gaston Bachelard

9.1 Function overloading

We can write a simple function to return the square of an integer value:

```
int square(int arg) {return (arg * arg);}
```

We can also write a function to return the square of a double with the same name!

```
double square(double arg) {return (arg * arg);}
```

In older languages, such as C and FORTRAN, it's impossible to have two functions with the same name. In C++, you can have as many functions as you like with the same name and this is called *function overloading*. To avoid confusion, functions with the same name should serve the same purpose, usually acting on different types of variables or classes. The one limitation is that C++ uses the arguments to the function to decide which version of the function to call. Thus, you can't have two functions with the same name and arguments, but different return types.

```
int solve_pde(double a);
float solve_pde(double a); // Illegal
int solve_pde(double a, int b) // OK
```

The compiler will determine which version of the function to use when it is called. The following code is illegal in C, but will compile and run in C++.

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

int square(int arg) {return (arg * arg);}
double square(double arg) {return (arg * arg);}
```

```
main()
 {
  int i=10;
  double a=20;

  cout << i <<  " squared is " << square(i) <<  "\n";
  cout << a <<  " squared is " << square(a) <<  "\n";
 }
```

9.1.1 Function Templates

A powerful generalisation of function overloading is provided by the function template, which allows data types to be parameters in function definitions. For example, a generic square function may be written:

```
template<class T> T square(T arg) {return (arg*arg);}
```

Here, the dummy class, `T`, denotes any valid class or data type and directly replaces `int` and `double` in the earlier definitions of the square function. The final program in §9.1 becomes

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

template<class T> T square(T arg) {return (arg*arg);}

main()
 {
  int i=10;
  double a=20;

  cout << i <<  " squared is " << square(i) <<  "\n";
  cout << a <<  " squared is " << square(a) <<  "\n";
 }
```

During the compliation process, the compiler recognises that `square` has been called with both `int` and `double` arguments and compiles the appropriate versions of `square`

from the templated definition. Templates may also be used in class definitions and in the development of generic algorithms, but, unfortunately, these exciting tricks will not be covered in this course.

9.2 Operator overloading

The overloading philosophy can be extended to operators, where it is used to define how the standard operators, e.g. `+ - * /` act on any classes you write. For example, we shall define a complex number class and overload the addition operator `+`.

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

//Let us allow the real and imaginary parts to be accessed outside class
class complex {
 public:
 double R; //real part
 double I; //imaginary part
 complex() {R = 0.0; I = 0.0;} //Basic constructor
 complex(double r, double i) {R = r; I = i;} //Overloaded Constructor
};

//Define overloaded addition
complex operator+(complex a, complex b)
 {
  //Just add real to real and imaginary to imaginary
  b.R += a.R;
  b.I += a.I;
  //Return new value
  return(b);
 }

main()
 {
  complex A(1.0,5.0), B(2.0,3.0), C;

  C = A + B;
```

```
  cout << A.R <<  " + " << A.I <<  "i" <<  " plus "
       << B.R <<  " + " << B.I <<  "i" <<  " = "
       << C.R <<  " + " << C.I <<  "i\n";
 }
```

As seen above, the `operator+` "function" is used to overload `+` with the two arguments being the variables on either side of the addition sign. The above code only redefines addition of two complex variables. If the following lines are introduced into `main`

```
complex A(1.0,5.0), C;
double b = 2.0;

C = A + b;
```

then the compiler complains:

```
test.cc:27: no match for ‘complex & + double &’
test.cc:14: candidates are: operator +(complex, complex)
```

The compiler has no function for the addition of complex and double and so one must be defined

```
//More overloaded complex addition
complex operator+(complex a, double b)
 {
  //Just add real to real part
  a.R += b;
  //Return new value
  return(a);
 }
```

Annoyingly, every possible combination of additions that are used must be defined. If we now write `C = b + A`, the compiler will again complain because we have only defined "complex" + "double", not "double" + "complex"!!

Every operator can be overloaded using the operator keyword, but some require more care than others. Particular care should be taken with the shortcut operators `+=`, etc.

9.2.1 Operator member functions and the this pointer

In §9.2 we used operator overloading functions just like normal functions. An alternative is to define them as member functions, which can be useful if they are to operate on private data. The only difference in the definition is that the first argument, the class itself, is implied. The complex addition operator would be:

```
 //Define overloaded addition, as a member
complex complex::operator+(complex b)
 {
  //Just add real to real and imaginary to imaginary
  b.R += R;
  b.I += I;
  //Return new value
  return(b);
 }
```

and `R` and `I` refer to the members of the current object (the object on the left of the addition sign).

In such functions, C++ defines an intrinsic pointer to the current object, the `this` pointer. For example, we could write `this->R` instead of `R`. In general, it is shorter and clearer not to use the `this` pointer, but what if we want to return the current object? In that case the return statement must be `return(*this);`, which is of use when overloading the `=` operator, e. g.

```
//Define overloaded equals (a member function)
//Must be defined as constant reference
complex complex::operator=(const complex &old_complex)
 {
 /Assign old values to current value
 R = old_complex.R;
 I = old_complex.I;
 //Return current object
 return(*this);
}
```

10 Inheritance

"The heathen are come into thine inheritance, And thy temple have they defiled."

— T. S. Eliot

C++ allows the creation of new, sometimes called *derived*, classes from existing, or *base* classes. For example, remember the student class from §6?

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

//Create a class student
class student {
  char name[80];
  double exam_result;
 public:
   void putname(char *n) {strcpy(name,n);}
   char *getname() {return name;}
   void putresult(double mark) {exam_result = mark;}
   double getresult() {return exam_result;}
 };
```

Now let's suppose that some of the students sit an extra exam. We could redefine the class, but instead we shall create a new class `keen_student`.

```
//Create derived class from the class student
class keen_student: public student {
 double exam2_result;
public:
 //Define function in body of class because it's short
 void putresult2(double mark) {exam2_result = mark;}
 double getresult2() {return exam2_result;}
};
```

The class `keen_student` contains all the variables and functions in the class `student` as well as those explicitly stated in the class definition. The new class can now be used exactly as any other class:

```
main()
```



```
  {
   student newton;
   keen_student einstein;

   newton.putname("Isaac Newton");
   newton.putresult(45.2);

   einstein.putname("Albert Einstein");
   einstein.putresult(50.0);
   einstein.putresult2(75.0);

   cout <<  "Exam results: " << "\n ";
   newton.getname(); cout  << " : " << newton.getresult() << "\n ";
   einstein.getname(); cout << " : " << einstein.getresult() <<  " , " <<
   einstein.getresult2() << "\n ";
  }
```

The general syntax for creating derived classes is as follows

`class class_name :` *access-specifier* `base_class_name {};`

The *access-specifier* can be **public**, **private** or **protected**, just as inside class definitions. The principle of encapsulation cannot be violated, and so the access-specifier only applies to the public and protected members of the base class. Private members of the base class always remain private to that class. Protected members of the base class can be used by derived classes, but cannot be used outside them. Finally, public members of the base class remain public in the derived class, unless overruled by the access-specifier. This can all get quite confusing, but the essence is that data can only be made "more" private by using an access-specifier in front of a class — private data can never be made public.

10.1 Multiple inheritance

A class may be derived from more than one base class, in which case the parent classes are separated by commas in the definition:

```
class derived: public base1, public base2 {};
```

A Multiple files

In large projects it is often convenient to split the source code into a number of separate files. Functions, classes and variables defined in one file cannot be used in another unless they are prototyped at the beginning of the file in which they are to be used. This is usually achieved via header files, which contain the prototypes. The standard convention is that if the source file is *file1.cc*, then the corresponding header file is *file1.h*.

```
//This is a file called square.cc
int square(int a)
 {
  return(a*a);
 }
//End of square.cc

//This is a file called square.h
int square(int a);
//End of square.h
```

The square function can now be used in any other program provided that the header file square.h is included.

```
//File called main.cc
 #include <iostream>
 #include  "square.h"
 using namespace std;

//Main function
main()
 {
  int i;

  for(i=1; i<=10; i++)
   {
   cout << i << " " << square(i) << "\n";
   }
 }
```


Another convention is that the names of local header files are enclosed in double quotes `"`, whereas standard header files are enclosed in angle brackets `<>`. In actual fact, either method can be used for any header file.

All the files containing source code for the whole program must now be compiled and linked together. The easiest method is just to put all the source files in the compilation command:

```
c++ main.cc square.cc
```

A.1 Makefiles

Once there are a large number of files, it is tedious to keep typing in the compilation command when you make changes to the source. **make** is a utility available on most platforms that will take care of this automatically. A special file which must be named *makefile* or *Makefile* contains the instructions telling make which files are included in each executable and how to compile it. A simple makefile is shown below:

```
# Simple makefile

program: square.h square.cc main.cc
	g++ -o program square.cc main.cc
```

The first line is a comment. The second establishes the name of a *target* **program** and the files that are needed to make that target: square.h, square.cc and main.cc. The third line must start with a TAB and then contains the command to compile the target from the source files. A makefile may contain many targets and their associated compilation commands. The target program is made by typing `make program` at the command line. In fact, just typing `make` will automatically make the first target in a makefile.

When make is invoked for a particular target it examines the timestamps of target and its source files. If any of the source files have been modified more recently than the target, the target is recompiled. Alternatively, if no file exists with the name target, then the compilation command is executed.