3D Game Programming using DirectX 10 and OpenGL's

Ancillary C++ eBook

By

Pierre Rautenbach

PREFACE

The Ancillary C++ eBook provides a systematic treatment of the C++ programming language. It lays down fundamental programming knowledge indispensable for any kind of software development. The ebook consists of a number of language specific programming topics, with each topic and section explained in a stepwise manner via code-based examples.

A Word on C++

C++ was designed by Bjarne Stroustrup while working at AT&T Bell Laboratories in the early 80s. C++ is the successor to the procedural C programming language (originally called *C with Classes*). Since its inception, C++ has become one of the most widely adopted languages, especially within the game and system programming industries. With procedural languages everything is executed as a sequence of instructions. C++ is an object orientated language. With object orientation everything is thought of as objects. All the instructions and procedures are seen as objects – alone standing entities that can be described as user created types. C++ is thus nothing more than an extension of C with the added ability to create objects and with additional features such as operator overloading, virtual functions, templates, exception handling and multiple inheritance.

C++, as covered in this ebook, conforms to the current ISO/IEC standardised version, specifically the ISO/IEC 14882:2003. A joint ANSI/ISO (International Standards Organisation) committee are responsible for this standardisation. The ANSI standard partitions C++ into two parts: the basic language and the C++ standard library. The C++ standard library consists of an adapted C standard library and the *Standard Template Library*, commonly referred to as the *STL* (a C++ library providing ready to use container classes, iterators and algorithms – all its components implemented as templates). This does not in any sense exclude the use of other libraries. The ISO/IEC standard is just in place to ensure a uniform C++ version. ANSI C++ code has the characteristic of being compilable on any ISO/IEC compliant compiler.

C++ is most definitely the choice of engine and game programmers. This can easily be substantiated by looking at the game industry or many of the freely/GPL licensed game engines available on the Internet – all of them written in C++.

eBook Features

Some of the key features of this ebook include:

- Thorough coverage of American National Standards Institute (ANSI) C++
- Concept of learning by example
- Figures and tables for the representation of properties and concepts
- Numerous cross-platform C++ examples

The reader will be guided though the material in a logical manner accompanied by constant notes and hints on common programming mistakes, concepts and trends.

Support Materials

All programming examples in this book are complete and all the related source files are available from the companion website: www.cengage.co.uk/rautenbach.

Source Compilation and Development Setup

Before you can compile and run the included examples, you'll need a compiler and/or alternatively an integrated development environment. There are several development environments available, both free and commercial. An excellent free development environment is *MinGW Developer Studio* (www.parinyasoft.com/). It is a cross-platform C/C++ IDE (Integrated Development Environment) for the GNU C/C++ compiler. On the commercial side there is Microsoft's highly acclaimed *Visual Studio* (http://msdn.microsoft.com/vstudio/). These development platforms are all-in-one solutions, each featuring an editor for writing the physical code, several tools useful during the development process (such as a debugger) and a compiler for building program executables. Alternatively you can always use a simple text editor such as *TextPad* (www.textpad.com/) or *Scite* (www.scintilla.org/SciTE.html) coupled with a C++ compiler. Such a setup will suffice for the development of a simple "Hello World" program to a full 40 000 line game engine. When choosing such a text editor, always choose one that supports a form of syntax highlighting.

The examples provided in this book are 100% cross platform and will compile on any ANSI compiler. All the C++ examples were tested on Linux and Windows XP/Vista via the MinGW compiler and Microsoft Visual Studio 2005.

Chapter 1

BASICS

1.1 Preview

Chapter 1 is of key importance as it deals with several basic C++ concepts. This chapter gives a brief tour of various C++ syntactic constructs (keywords) including some more advanced topics such as pointer and reference operations. All topics are presented in an example driven fashion. Chapter 1 also focuses on additional C++ syntax, variables, expressions, operator precedence and arithmetic operators.

1.2 Your First C++ Program

Since the early days of programming, whenever using a new language for the first time, it has been customary to print the string “Hello World”. Printing a text message to the screen is generally a syntactically simple yet purposeful way of introducing a new programming language. Here is a C++ “Hello World” program:

Program 1.1 – The Hello World tradition

```
/*
=================
HelloWorld.cpp
- My very first program in C++
=================
*/

#include <iostream>
using namespace std; //just ignore it for the moment

int main()
{
	cout << "Hello World!" << endl; //prints the string Hello World
	return 0;
}
```

Comments `//   /*…*/`

Comments are textual lines ignored by the compiler. Their purpose, as suggested by their name, is to add meaning to a portion of written code. This might seem unnecessary in a way, however, just imagine going through a thousand lines of uncommented code written by someone else and making sense of it all. Comments are thus crucial for summarising code. They are also an excellent way for keeping track of progress and to highlight unfinished or buggy sections in a program.

There are two types of comments, the first and oldest is signified by the set: (‘`/* */`’). These are called *multi-line* comments. Everything enclosed between them is ignored and normally marked green by the code editor. *Multi-line* comments work as a pair, in other words, a comment of this type will always start with a *slash* (‘`/`’) followed by a *star* (‘`*`’) and the actual comment. Termination of the comment is signalled with a *star-slash* (‘`*/`’).

The second kind of comment is the *double slash* (‘`//`’). This is the C++ style comment, the multi-line comment (‘`/* */`’) is the old C style; remember, everything supported in C is supported in C++. The double slash (‘`//`’) only works for single line remarks.

Include `#include`

The include directive (‘`#include`’) is used for the inclusion of header files. This time I included ‘`iostream.h`’. The hash sign (‘#’) signals the C++ preprocessor (active during the compilation process) to include the specified files. The *preprocessor* is a program run during the compilation cycle and it is responsible for the transformation of source code prior to the actual compilation step. Header files are saved with a *dot-h* (‘`.h`’) file extension. Thus, to summarise, the include directive (‘`#include`’) is used whenever another source or header file is to be included by the program. Just a note on the angular brackets (‘< >’) following the include directive; whenever the preprocessor reads an angular bracket (‘< >’) it looks for the given file in the compiler’s default or user specified include directories. Custom written header files are included with *double-quotes* (‘“ ”’).

Without the inclusion of ‘`iostream`’ you would not be able to use the standard output stream object (‘`cout`’). The ‘`cout`’ object provides command-line text output to C++ programs; don’t expect to see it that much outside the terminal environment though.

NOTE: Writing ‘`#include <iostream.h>`’ is perfectly legal, however, the dot-h (‘`.h`’) file extension for default headers has been deprecated in C++.

NOTE: In C one would use ’`#include <stdio.h>`’ instead of ‘`#include <iostream.h>`’. Standard textual output, such as Program 1.1’s will then be generated via the statement ‘`printf(“Hello World!\n”)`’ instead of ‘`cout << “Hello World!” << endl`’.

NOTE: The two most frequently used preprocessor commands are '`#include`' and '`#pragma`'.

Main `int main()`

The entry point of Program 1.1 is given on line 9. This entry point, `main()`, is a *function* required by all C++ programs. A *function* is just a portion of code performing some action or task (Program 1.5 contains several functions). The main function is always called, no matter what.

Braces `{ }`

Braces ('`{}`') indicate the start and end of a code-block. The *left-brace* ('`{`') denotes a *Begin*, with the *right-brace* ('`}`') indicating an *End*. Everything contained within these braces is considered part of whatever lies prior to the first brace. All functions must have them, even if the function is empty.

Direction operators `>>    <<`

The Line, '`cout << "Hello World!" << endl`', is the functional unit of this program, without it program 1.1 would not do anything. The output redirection operator ('<<') is used, as its name suggests, to direct output. In this case it directs output to the screen. There can be quite a number of these direction operators in such a '`cout`' statement; for example, the second direction operator writes a new-line to the screen. The endline ('`endl`') statement is similar to C's newline escape sequence ('\n') – both of them indicating a line break. Omitting the endline iostream manipulator will cause the next line of text to be placed right next to this "`Hello World`" string.

Return types `return 0;`

A final thought on program 1.1; it would also have worked if I used '`void main()`' instead of '`int main()`'. The '`void`' type signals the return of nothing, where '`int`' indicates the return of an integer – in this case a zero '`0`'. Programmers refer to '`void`' and '`int`' as *return types*. The compiler expects something to be returned whenever a return type is placed in front of a function name (it generates a compiler error when the expected type isn't returned). Program 1.1's main function returns the value zero ('`return 0`'). Using '`void`' as a return type means no '`return`' statement, however, the ANSI C++ standard forbids this. All ANSI compliant main functions should return an ‘`int`’.

Compilation and Output

To compile the program, execute the following from within its file location:

```
g++ -c HelloWorld.cpp
g++ HelloWorld.o –o chosen_executable_name
```

To run the program type:

```
chosen_executable_name
```

It will output the following to the screen:

```
Hello World!
```

NOTE: There are various different types of error messages. *Compile-time errors* are errors picked up by the compiler during source compilation. *Runtime errors* occur during program execution – they are caused by programming bugs, for example.

Example of a runtime error: running out of storage space in an array.

Example of a compile-time error: writing ‘`imt main()`’ instead of ‘`int main()`’. There are of course hundreds of ways to cause compile-time errors, anything from assigning to comparing variables of different types.

Good, there you go, your very first C++ program. Easy isn’t it? Of course! Just a note on indentation – it is completely dependant on the programmer’s preference and will define his/her very own coding style. Just remember that someone else might not be able to understand a program if its coding style isn’t clear. Also remember, C++ is a case sensitive language: `Main` is not the same as `main` and will generate an error.

1.3 Basics:

Before discussing the next example, let’s have to look at some C++ basics. Basics refer to the actual language syntax – mainly the C++ alphabet utilised to make words; i.e.

programs and ultimately games. Once you know the words you can make sentences, paragraphs and so on. If you are unfamiliar with programming in general, it is extremely important to thoroughly understand this section before moving on to the next. Variables are the basis of pointer and reference operations (covered in section 1.4). The focus now will be on variable types, expressions, operator precedence as well as assignment and increment/decrement operators.

1.3.1 Variables

Variables can be seen as fixed or dynamically creatable storage space. They give a program the ability to store and manipulate data. Variables have a *name* and a *type*. The type tells the variable what kind of data to store with its name serving as identification. A variable of type *string*, for example, cannot hold an *integer* value and vice versa.

You *define* a variable like this:

```
int numberOfKids;
```

This variable can also be *initialised* with a value, either when defining it:

```
int numberOfKids = 2;
```

or at a later stage in your program:

```
numberOfKids = 2;
```

The variable ‘`numberOfKids`’ is of type ‘`int`’, it can thus only contain integer values – nothing else.

You can define more than one variable in a single line:

```
int numberOfKids, numberOfDogs;
```

The above defined variables, ‘`numberOfKids`’ and ‘`numberOfDogs`’, have the same type, namely, ‘`int`’.

You can also initialise one or both of these variables:

```
int numberOfKids = 1, numberOfDogs;
```

By doing this, I have given ‘`numberOfKids`’ the default value ‘1’ with ‘`numberOfDogs`’ left *uninitialised*.

NOTE: You can use any variable name as long as it is not the same as a standard C++ keyword (`main`, `void`, `typedef`, `struct`, `class`, etc).

NOTE: Don’t go crazy when naming your variables. Using variable names such as ‘`p434U2R`’ is a very bad idea. Stick to clear descriptive variable names for the sake of your sanity.

NOTE: Every programmer out there has a coding style. This style is developed over time. There are also standard styles like the Hungarian notation (used by big companies like Microsoft, for example).

Table 1.1 lists the most common variable types with their corresponding sizes in bits (one byte equals eight bits) – computer memory is grouped into storage holes, each with a physical size of 1 byte.

Table 1.1 – The main variable types available

Variable Type	Name	Value Range
int	Integer (32bit)	-2 147 483 648 to 2 147 483 647
short	Short Integer (16bit int)	-32 768 to 32 767
long	Long Integer (32bit int)	-2 147 483 648 to 2 147 483 647
char	Character (8bit)	256 characters
float	Floating point (32bit real)	1.2exp38 to 3.4exp38
double	Double float (64bit real)	2.2exp308 to 1.8exp308
unsigned int	Unsigned Integer (32bit)	0 to 4 294 967 295
unsigned short	Unsigned Short Integer	0 to 65 535
unsigned long	Unsigned Long Integer	0 to 4 294 967 295
bool	Boolean	True or False

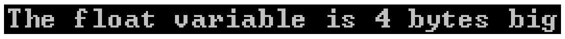
NOTE: Unsigned variables can only hold positive values; their value range starts at zero but the variable’s absolute size stays the same as that of its signed counterpart.

NOTE: Very old compilers and computers represented the ‘`int`’ type as a 16bit value; it thus had the same value range as today’s ‘`short`’ variable type. To be on the safe side, always use ‘`long`’ when referring to a 32bit ‘`int`’ and ‘`short`’ when referring to a 16bit one (not doing so might lead to strange results when running your code on different machines).

You can use the ‘`sizeof()`’ function in C++ to get the size of a specific object. For example, the following line of code will return the size of the float variable in bytes:

```
cout << “The float variable is ” << sizeof(float) << “ bytes big”
<< endl;
```

This line of code generates the following output:



```
The float variable is 4 bytes big
```

1.3.2 Operators

Operators are a critical component of any programming language seeing as they are responsible for the programming language’s basic arithmetic functionality. They also have a specific precedence and evaluation order.

Table 1.2 gives a list of C++ operators along with their order of precedence. *Precedence* is order of evaluation. For example, calculating 5*8+7*2+1 yields (40)+7+(14)+1 which in turn equates to 62.

Table 1.2 – Operators (in order of precedence)

Operator	Description
()	Brackets
*, /, %,	Multiply, Divide, Modulo
+,-	Plus, Minus
<, <=, >, >=	Smaller than, Smaller or Equal to, Bigger than, Bigger or Equal to
==, !=	Equals, Not Equals
=	Assignment operator

NOTE: Modulo (%) is used to calculate the remainder of integer division. For example; 17%5 evaluates to 2. Mod is very useful in looping structures (if something has to be executed every 10th time, you simply execute where x%10 equals 0).

Table 1.3 gives a list of assignment operators and Table 1.4 lists some increment/decrement operators.

Table 1.3 – Assignment operators

Operator	Example	Equivalent
%=	varName %= 10;	varName = varName % 10;
*=	varName *= 3;	varName = varName * 3;
/=	varName /= 2;	varName = varName / 2;
+=	varName += 9;	varName = varName + 9;
-=	varName -= 5;	varName = varName - 5;

Table 1.4 – Increment/Decrement operators

Operator	Example	Equivalent
++ (pre)	++varName;	varName = varName +1; (increment then use)
++ (post)	varName++;	varName = varName +1; (use then increment)
-- (pre)	--varName;	varName = varName - 1; (increment then use)
-- (post)	varName--;	varName = varName - 1; (use then increment)

Let's continue with the next example. The basic idea behind this program is to illustrate variable and operator use. The given program layout is similar to that of Program 1.1.

Program 1.2 – Variables & Expressions

```
/*
=================
Variables.cpp
- A program demonstrating the use of variables
=================
*/

#include <iostream>

int main()
{
        //variable definitions & some default initializations
        int numberOfKids = 2, numberOfDogs;

        //main program code
        cout << "First printout" << endl;
        //2 endl below so there is a line open before second printout
        cout << "Number of kids: " << numberOfKids << endl << endl;

        numberOfDogs = 4;
        cout << "Second printout" << endl;
        cout << "Number of kids: " << numberOfKids << endl;
        cout << "Number of dogs set to 4: " << numberOfDogs << endl << endl;

        numberOfDogs = 0;
        numberOfKids = numberOfKids + 1; //same as numberOfKids++
22          cout << "Third printout" << endl;
23          cout << "Number of kids with added 1 kid: " << numberOfKids << endl;
24          cout << "Number of dogs set to 0: " << numberOfDogs << endl << endl;

25          numberOfDogs = numberOfKids;
26          numberOfKids += 2; //same as numberOfKids = numberOfKids + 2;
27          cout << "Fourth printout" << endl;
28          cout << "Number of kids with added 2 kids: " << numberOfKids << endl;
29          cout << "Number of dogs set to the previous number of kids: " <<
    numberOfDogs << endl << endl;

30          cout << "Fith printout" << endl;
31          cout << "Number of kids - 3: " << numberOfKids-3 << endl;
32          cout << "Number of dogs * 2: " << numberOfDogs*2 << endl;

33          return 0;
34  }
```

Definitions and Initialisations

On line 11 I define two variables, ‘`numberOfKids`’ and ‘`numberOfDogs`’. Both these variables are of type integer.

On line 16 I assign the value ‘4’ to ‘`numberOfDogs`’. On line 20 I assign the value ‘0’ to it and on line 25 I assign the value currently contained within the ‘`numberOfKids`’ variable to it.

Looking at line 32 you’ll see some arithmetic being performed on the variable ‘`numberOfDogs`’. Writing ‘`numberOfDogs*2`’ in the ‘`cout`’ line multiplies the current value contained within the variable by ‘2’. Following this calculation, I output the result.

Similar operations can be observed when looking at the ‘`numberOfKids`’ variable. I just want to highlight line 26 – notice the *increment* statement:

```
numberOfKids += 2;
```

This is exactly the same as writing:

```
numberOfKids = numberOfKids + 2;
```

The same thing can be observed when looking at line 21; the results obtained by writing:

```
numberOfKids = numberOfKids + 1;
```

can also be obtained from the statement:

```
numberOfKids++;
```

Writing the *increment operator* ('++') after the variable name tells the compiler to first use the current value stored in the variable, then increment it. In our example's case it doesn't matter whether you write:

```
++numberOfKids;
```

or

```
numberOfKids++;
```

both will have the same effect. An increment operator in front of the variable name is called a *prefix increment operator* while the opposite is called a *postfix increment operator*.

> **NOTE:** The assignment ("=") operator is one of the few operators read from right to left. Such an operator is called right-associative. *Associativity* is a mathematical property held by a binary operation.

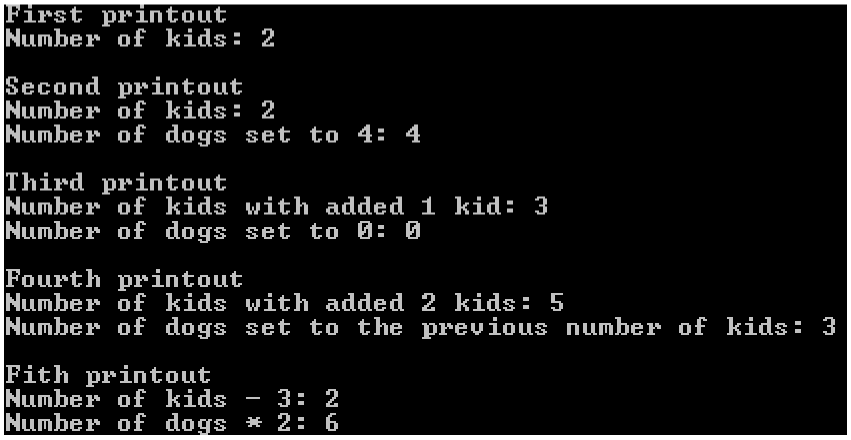
Compilation and Output

To compile the program type the following from within its directory:

```
g++ -c Variables.cpp
g++ Variables.o –o your_name
```

To run the program type:

```
your_name
```



```
First printout
Number of kids: 2

Second printout
Number of kids: 2
Number of dogs set to 4: 4

Third printout
Number of kids with added 1 kid: 3
Number of dogs set to 0: 0

Fourth printout
Number of kids with added 2 kids: 5
Number of dogs set to the previous number of kids: 3

Fith printout
Number of kids - 3: 2
Number of dogs * 2: 6
```

1.4 Values by Reference:

Program 1.2 illustrates the storage of values by means of variables for the purpose of manipulation and/or the accessing of these values at a later stage. Each of these values; stored in physical memory locations or cells, are accessible by means of a specific variable name. This frees us of the technicalities associated with locating data in physical memory. In program 1.2 I simply call the variable identifier whenever I want to access the stored data.

Suppose you have a program with two integer variables, '`int1`' and '`int2`'. Now, assigning values to these variables ('`int1 = 20`', '`int2 = 7`') will cause both values to be written into physical memory. Figure 1.1 shows the memory locations after this assignment.

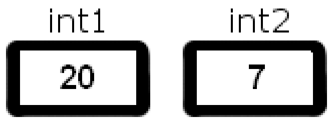


Fig 1.1 Variables stored in memory

1.4.1 Pointers

Pointers are variables that store the memory addresses of other variables. They basically give programs the ability to manipulate data stored in other variables. Just like variables, pointers also have a name and type.

The address of a variable refers to the location of the physical memory allocated to that specific variable. This address is like a tag that links the variable to a physical location in memory.

A pointer can be defined as follows:

```
int *numberOfKids;
```

NOTE: All pointers are defined with an asterisk ('*').

Pointers can be initialised to a physical memory address. It is also safe programming practice, when defining a pointer, to initialise it to zero '0' or NULL. *Null* is a symbolic constant used in C and C++. A pointer initialised to NULL points to nothing.

Initialisation of a pointer is done in exactly the same way as with variables; after all, it is just a different kind of variable. Only addresses should be assigned to pointers.

NOTE: No integer, string, char or any other type may directly be assigned to a pointer.

So why use pointers at all? If pointers can only be assigned addresses, how do you make them point to the address of a variable to begin with? Easy! There is another operator called the *address operator* ('`&`').

Say you have an integer pointer, '`myVarPtr`', and an integer variable, '`myVar`'. Let's first define and initialise them:

```
int *myVarPtr = NULL; //myVarPtr points to an object of type int
int myVar = 8; //initialise myVar to 8
```

'`myVarPtr`' points to an object of type '`int`' and is initialised to NULL while '`myVar`', an integer variable, is assigned the value '8'.

Now I want '`myVarPtr`' to point to the address of '`myVar`', this is achieved by writing:

```
myVarPtr = &myVar;
```

That's it, '`myVarPtr`' now points to the address of '`myVar`'.

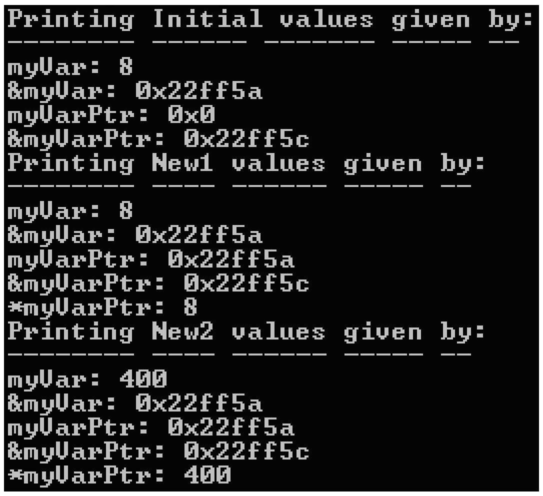
You should use the indirection operator ('`*`') to get the value pointed to by '`myVarPtr`'. For example: '`cout << *myVarPtr;`' will print the number '8'.

The following program illustrates the basics of pointer operations:

Program 1.3 – Pointer operations

```
/*
=================
Pointers.cpp
- A program that performs pointer operations
=================
*/

#include<iostream>

using namespace std; //ignore for now, will explain later

/*
=================
main function
- main program
=================
*/
int main()
{
        short *myVarPtr = NULL;
        short myVar = 8;

        cout << "Printing Initial values given by: " << endl;
        cout << "-------- ------ ------- ----- --" << endl;
        cout << "myVar: " << myVar << endl; //gives the value of variable
        cout << "&myVar: " << &myVar << endl; //gives address of variable in memory
//      cout << "*myVar: " << *myVar << endl; COMPILE ERROR - NOT A POINTER
        cout << "myVarPtr: " << myVarPtr << endl; //points to nothing, gives address to what it points
        cout << "&myVarPtr: " << &myVarPtr << endl; //gives address of pointer in memory
//      cout << "*myVarPtr: " << *myVarPtr << endl; RUN TIME ERROR - DOESN'T POINT TO ANYTHING

        myVarPtr = &myVar; //to point to another variable

        cout << "Printing New1 values given by: " << endl;
        cout << "-------- ---- ------ ----- --" << endl;
        cout << "myVar: " << myVar << endl; //gives the value of variable
        cout << "&myVar: " << &myVar << endl; //gives address of variable in memory
        cout << "myVarPtr: " << myVarPtr << endl; //gives address to what it points
        cout << "&myVarPtr: " << &myVarPtr << endl; //gives address of pointer in memory
        cout << "*myVarPtr: " << *myVarPtr << endl; //gives the value to what it points

        *myVarPtr = 400; //to assign a value

        cout << "Printing New2 values given by: " << endl;
        cout << "-------- ---- ------ ----- --" << endl;
        cout << "myVar: " << myVar << endl;
        cout << "&myVar: " << &myVar << endl;
        cout << "myVarPtr: " << myVarPtr << endl;
        cout << "&myVarPtr: " << &myVarPtr << endl;
        cout << "*myVarPtr: " << *myVarPtr << endl;

        return 0;
}
```

Output



```
Printing Initial values given by:
-------- ------- ------- ----- --
myVar: 8
&myVar: 0x22ff5a
myVarPtr: 0x0
&myVarPtr: 0x22ff5c
Printing New1 values given by:
-------- ---- ------ ----- --
myVar: 8
&myVar: 0x22ff5a
myVarPtr: 0x22ff5a
&myVarPtr: 0x22ff5c
*myVarPtr: 8
Printing New2 values given by:
-------- ---- ------ ----- --
myVar: 400
&myVar: 0x22ff5a
myVarPtr: 0x22ff5a
&myVarPtr: 0x22ff5c
*myVarPtr: 400
```

Pointing to memory

The comments given in the source should explain everything there is to this little program. It is basically a review of what has already been said about pointers.

I would, however, like to highlight the code from line 27 to 37. Line 29 assigns the address of ‘`myVar`’ to the pointer ‘`myVarPtr`’. As can be seen from the output, ‘`myVar`’ has a memory address of ‘`0x22ff5a`’ with ‘`myVarPtr`’ assigned the address ‘`0x22ff5c`’. This address assignment is system dependent and both will be different because both variables exist in different parts of system memory. A pointer’s address remains constant during its entire lifespan.

Have a look at the last part of the program. The statement:

```
*myVarPtr = 400;
```

on line 37 sets the value pointed to, to ‘400’. This causes the value contained within ‘myVar’ to be set to ‘400’.

It is very important not to become confused with the address held by a pointer and the corresponding value at that address. Pointers can be summarised as follows:

`myVarPtr` //gives the address of what the pointer points to
`&myVarPtr` //gives the address of the pointer itself in memory
`*myVarPtr` //gives the value of what the pointer points to

So why use Pointers? Aren't variables enough? The answer to the latter question is shortly no. But unfortunately the usefulness of pointers will not become clear until you start accessing function and class members. Say you want to send data from one part in your program to another – which of the following seems better, sending the actual value or sending a reference to that value? Remember, our ultimate goal is to do state of the art 3D programming; speed is one of the biggest issues when doing that and using pointers saves a bit of time every time! Their use will become crystal clear during the sections on Functions and Classes.

The final object to mention is the *delete* operator. As seen in program 1.3; a pointer exists in the same part of memory, '`0x22ff5c`' in our case, from the moment it's defined until program termination. To free up this memory location, you'll need to make use of the delete operator at the end of your program. It can be done in the following manner:

```
delete myVarPtr;
```

1.4.2 References

There really isn't that much of a difference between pointers and references. A reference can be seen as an alias (an alternate name for a variable/object). References are frequently used with parameter passing and when returning values (fully explained during the section on functions). References have similar capabilities when compared to pointers but the syntax used is much more straightforward.

One thing to remember when defining references is that a reference must be initialised while being defined. Just as important; once you have initialised a reference you cannot change it.

In program 1.3's context you can *define and initialise* a reference by writing:

```
int &myVarRef = myVar;
```

NOTE: A reference is defined with an ampersand sign ('&') in front of its name.

NOTE: The address of the bitwise AND operator ('&') used with variables is not the same as the reference operator ('&'). The symbol used to represent them is the only thing related.

A reference is an alias; hence, everything that happens to the reference happens to the referenced object. You can protect an object from being changed in such a way by declaring the reference as constant (using the `const` keyword). This adds a very interesting attribute to the reference. You can only initialise a reference with a value if it's a constant reference. For example:

`const int &myVarRef = 9;` //the value that it is a reference to cannot change

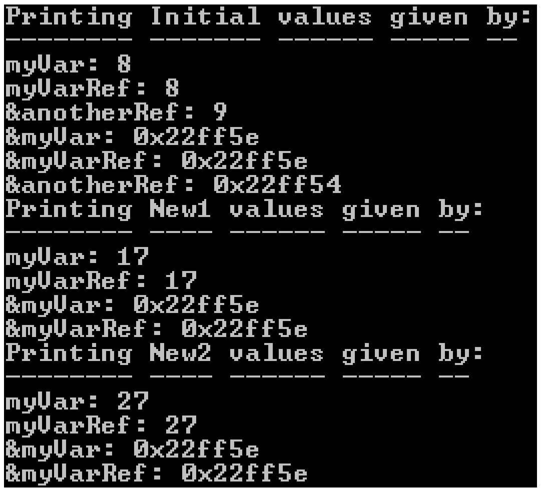
Program 1.4 takes a closer look at reference operations:

Program 1.4 – Reference operations

```
/*
=================
References.cpp
- A program that performs reference operations
=================
*/

#include<iostream>

using namespace std;

/*
=================
main function
- main program
=================
*/
int main()
{
        short myVar = 8;
        short &myVarRef = myVar;
        const int &anotherRef = 9; //only const can be initialised with value

        cout << "Printing Initial values given by: " << endl;
        cout << "-------- ------- ------ ----- --" << endl;
        cout << "myVar: " << myVar << endl; //gives value of myVar
        cout << "myVarRef: " << myVarRef << endl; //gives value of myVar
        cout << "&anotherRef: " << anotherRef << endl; //gives value of anotherRef
        cout << "&myVar: " << &myVar << endl; //gives memory address of myVar
        cout << "&myVarRef: " << &myVarRef << endl; //gives memory address of myVarRef
        cout << "&anotherRef: " << &anotherRef << endl; //gives memory address of anotherRef

        myVar = 17; //myVar is assigned the integer 17

        cout << "Printing New1 values given by: " << endl;
        cout << "-------- ---- ------ ----- --" << endl;
        cout << "myVar: " << myVar << endl;
        cout << "myVarRef: " << myVarRef << endl;
        cout << "&myVar: " << &myVar << endl;
        cout << "&myVarRef: " << &myVarRef << endl;
```

```
            myVarRef = 27; //myVarRef hence myVar is assigned 27
//          anotherRef = 7; //ERROR - const, the value cannot be changed

            cout << "Printing New2 values given by: " << endl;
            cout << "-------- ---- ------ ----- --" << endl;
            cout << "myVar: " << myVar << endl;
            cout << "myVarRef: " << myVarRef << endl;
            cout << "&myVar: " << &myVar << endl;
            cout << "&myVarRef: " << &myVarRef << endl;

            return 0;
}
```

Output



```
Printing Initial values given by:
-------- ------- ------ ----- --
myVar: 8
myVarRef: 8
&anotherRef: 9
&myVar: 0x22ff5e
&myVarRef: 0x22ff5e
&anotherRef: 0x22ff54
Printing New1 values given by:
-------- ---- ------ ----- --
myVar: 17
myVarRef: 17
&myVar: 0x22ff5e
&myVarRef: 0x22ff5e
Printing New2 values given by:
-------- ---- ------ ----- --
myVar: 27
myVarRef: 27
&myVar: 0x22ff5e
&myVarRef: 0x22ff5e
```

In this chapter you were introduced to many of the basic C++ language features. From standard program layout, data and text output to user input, arithmetic calculations and even advanced concepts such as pointers and references. Chapter 2 builds on these concepts, introducing structured programming, flow of control (concerned with the execution order of statements) and iteration.

Chapter 2
FLOW OF CONTROL AND ITERATION

2.1 Preview

Any computer program consists of several instructions. These instructions are by default executed in a linear manner (line by line). There are, however, ways to change this standard flow of control. For example, one method is to suddenly invoke a function. This will cause the program to jump to and work through the function, returning to its previous position only after successful execution of the function.

Something useful to remember is that all programs can be written in terms of:

- Sequential structures (one instruction is executed after the other)
- Selection structures (if, if/else, switch)
- Looping structures (for, while, do/while)

Chapter 2 discusses each of these structures in detail.

2.2 The goto statement

The most basic way of interrupting sequential program flow is to use the ‘`goto`’ statement. It should, however, be noted that the indiscriminate use of ‘`goto`’ is a really bad idea. Research done by Bohm and Jacopini in 1966 demonstrated that programs could be written without ‘`goto`’ statements. At this time the notion of structured programming came into effect, becoming synonymous with ‘`goto`’ elimination.

The ‘`goto`’ statement makes use of *labels* to direct flow. A label is placed left to a ‘`goto`’ statement, followed by a colon (‘`:`’) after its name. Thus, flow of control is immediately directed to the labelled position when a ‘`goto`’ statement is encountered. For example:

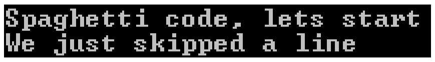
```
yourlabelname: number = number + 1;
goto yourlabelname; //jumps back to previous line
```

The following program illustrates the use of the ‘`goto`’ statement.

Program 2.1 – goto

```
1   /*
2   =================
3   Spaghetti.cpp
4   - A program illustrating the use of the goto statement
5   =================
6   */

7   #include<iostream>

8   /*
9   =================
10  main function
11  - main program
12  =================
13  */
14  int main()
15  {
16              cout << "Spaghetti code, lets start" << endl;
17              goto looplabel;

18              cout << "This line of code is skipped completely" << endl;

19    looplabel: cout << "We just skipped a line" << endl;

20              return 0;
21  }
```

Output



```
Spaghetti code, lets start
We just skipped a line
```

Spaghetti wise

The given program starts by printing the text as specified on line 16. It then progresses sequentially to line 17 where flow of control is interrupted – the program jumping straight to the label on line 19 and skipping the code on line 18 completely.

2.3 The if statement

The ‘`if`’ statement allows a program to carry out different actions depending on the validity of a test condition (for example: are two variables equal, if they are do the following…).

A basic ‘`if`’ statement looks this:

```
if(number1 == number2)
```

```
     number3 *= number1;
```

The ‘`if`’ statement is composed of two parts: the test condition and the body. The body is only executed if the test is true. If the condition has a value of ‘0’, it is considered false and the body is skipped. Any non-zero value is considered true, thus resulting in the execution of the body.

Table 1.2 gives a list of operators for use with ‘`if`’ statement conditional testing.

NOTE: Do not confuse the assignment “=” operator with the equals operator “==”.

Remember, you can also block statements together within the body of an ‘`if`’ statement. Doing so will lead to the execution of more than one statement.

An ‘`if`’ statement can also test for more than one conditionals at a time, this is done via the grouping of logical operators.

Logical operators (Boolean operators) evaluate to either TRUE or FALSE; examples include:

!	-	Logical NOT
&&	-	Logical AND
||	-	Logical OR

The following ‘`if`’ statement combines two test conditions by means of the logical AND operator:

```
if(number1 == 1 && number2 == 1)
     number3 *= number1;
```

The body of this loop is only executed if both numbers are equal to ‘1’.

The next ‘`if`’ statement combines two test conditions via a logical OR operator:

```
if(number1 == 1 || number2 == 1)
     number3 *= number1;
```

The body of this loop is executed if either one of the numbers or both are equal to ‘1’.

The following ‘`if`’ statement uses the logical NOT operator:

```
if(number1 != 1)
```

```
     number3 *= number1;
```

The body of this loop is executed unless the test condition equals ‘1’.

If statements can also be nested (the second if statement is only executed if the first one evaluates to true):

```
if(number1 != 1)
     if(number2 >= 17)
          number3 *= number1;
```

As mentioned previously; all TRUE values evaluate to ‘1’ whereas all FALSE values are evaluated to ‘0’. Any nonzero number is thus TRUE. The following examples illustrate this:

NOT (!)

A	!A
0	1
Nonzero	0

AND (&&)

A	**B**	**A&&B**
0	0	0
0	Nonzero	0
Nonzero	0	0
Nonzero	Nonzero	1

OR (| |)

A	**B**	**A| |B**
0	0	0
0	Nonzero	1
Nonzero	0	0
Nonzero	Nonzero	1

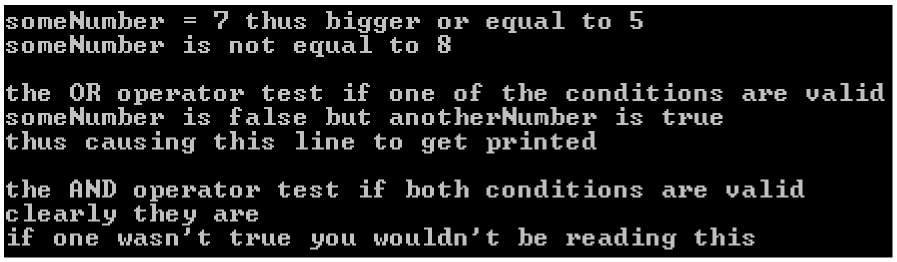
Program 2.2 plays around with different ‘`if`’ statement configurations. It clearly illustrates the practical implementation of the discussed concepts.

Program 2.2 – Variations on if

```
/*
=================
TheIfStatement.cpp
- A program showing various if statements
=================
*/

#include<iostream>
```

```
/*
=================
main function
- main program
=================
*/
int main()
{
        int someNumber = 7;
        short anotherNumber = 8;

        //basic if
        if(someNumber >= 5)
                cout << endl << "someNumber = " << someNumber << " thus bigger or equal to 5" << endl;

        //basic if
        if(someNumber != 8)
                cout << "someNumber is not equal to 8" << endl << endl;

        //the OR (||) operator
        if((someNumber != 7)||(anotherNumber == 8))
        {
                cout << "the OR operator test if one of the conditions are valid " << endl;
                cout << "someNumber is false but anotherNumber is true " << endl;
                cout << "thus causing this line to get printed" << endl << endl;
        }

        //the AND (&&) operator
        if((someNumber == 7)&&(anotherNumber == 8))
        {
                cout << "the AND operator test if both conditions are valid " << endl;
                cout << "clearly they are " << endl;
                cout << "if one wasn't true you wouldn't be reading this" << endl << endl;
        }

        //the AND (&&) operator with a NOT (!)    -          NOT PRINTED
        if((someNumber == 7)&&(anotherNumber != 8))
        {
                cout << "the AND operator test if both conditions are valid " << endl;
                cout << "clearly they are not since anotherNumber is in fact 8 " << endl;
                cout << "THIS LINE ISN'T PRINTED" << endl << endl;
        }

        return 0;
}
```

Output



```
someNumber = 7 thus bigger or equal to 5
someNumber is not equal to 8

the OR operator test if one of the conditions are valid
someNumber is false but anotherNumber is true
thus causing this line to get printed

the AND operator test if both conditions are valid
clearly they are
if one wasn't true you wouldn't be reading this
```

2.4 The if/else statement

The 'if' statement is used to control whether a statement is executed or not. Coupling it with 'else' allows you to execute something else if your original test condition wasn't met.

An 'if/else' statement has the following basic structure:

```
if(number1 == number2)
     number3 *= number1;
else
     number1 = number2;
```

2.5 The switch Multiple-Selection Structure

The 'switch' multiple-selection structure can be considered the 'if' statement's big brother. The main difference between them is that where 'if' evaluates a single value, 'switch' allows for the branching on several different values. 'switch' structures normally consist of several 'case' labels, 'break' statements and one 'default' case.

A basic 'switch' structure is illustrated here:

```
switch(userInput)
{
     case 1:
          //some statement
     break;

     case 2:
     {    //grouping can be done!
          //some statement
          //another statement
     }
     break;

     case 3 : case 4:
          /*more than one statement can be concurrently
          tested*/
     break;

     default: //handles anything else entered
          cout << "Incorrect choice, enter
                         choice again" << endl;
     break;
}
```

The first ‘`case`’ matching the given ‘`switch`’ value is executed. Execution stops at the first ‘`break`’. The next ‘`case`’ will also be executed if the subsequent ‘`break`’ statement is missing. The ‘`default`’ case can be left out but its inclusion generally ensures the handling of incorrect input.

NOTE: The switch structure can only be used in the testing of integers and characters.

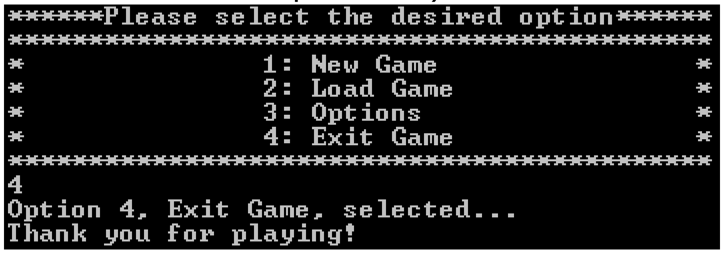
The following program creates a basic selection menu. The ‘`switch`’ statement tests the user’s input and prints a message according to the option selected.

Program 2.3 – An example using switch

```
/*
=================
Switch.cpp
- A program using the switch statement to see what menu option the user selected
=================
*/

#include<iostream>

/*
=================
main function
- main program
=================
*/
int main()
{
          short userInput;

          //menu printed to screen
          cout << endl << "******Please select the desired option******" << endl;
```

```
    cout << "********************************************" << endl;
    cout << "*              1: New Game              *" << endl;
    cout << "*              2: Load Game             *" << endl;
    cout << "*              3: Options             *" << endl;
    cout << "*              4: Exit Game             *" << endl;
    cout << "********************************************" << endl;

    cin >> userInput; //variable used to store the users input (controlling expression)

    //the switch selection structure testing the controlling expression
    switch(userInput) //the value of the expression is compared with each case label
    {
        case 1:
            cout << "Option 1, New Game, selected..." << endl;
        break;

        case 2:
            cout << "Option 2, Load Game, selected..." << endl;
        break;

        case 3:
            cout << "Option 3, Options, selected..." << endl;
        break;

        case 4:
        {
            cout << "Option 4, Exit Game, selected..." << endl;
            cout << "Thank you for playing!" << endl;
        }
        break;

        case '\n': case' ': //ignore a newline(ENTER) or space as input
            break;

        default: //handles anything else entered
            cout << "Incorrect choice, enter choice again" << endl;
        break;
    }

    return 0;
}
```

Output (on selection of option 4)



```
******Please select the desired option******
********************************************
*               1: New Game                *
*               2: Load Game               *
*               3: Options                 *
*               4: Exit Game               *
********************************************
4
Option 4, Exit Game, selected...
Thank you for playing!
```

NOTE: It’s a good idea to add a case statement testing for newline (‘\n’) and space characters (‘ ‘).

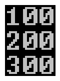
2.6 The while loop

Looping is an essential part of most programs. A group of instructions executed repeatedly are referred to as a loop. A ‘`while`’ loop is a repetition structure used to repeat a sequence of statements or actions. This sequence of statements is repeated infinitely, or until the starting condition becomes false.

Program 2.4 – An example for using while

```
1   /*
2   =================
3   While.cpp
4   - A program using the while statement, also features the use of the 'mod' operator
5   =================
6   */

7   #include<iostream>

8   using namespace std;

9   int main()
10  {
11          int k, i; //two counters

12          k = 1; //initialise k to 1

13          while(k++ <= 300) //increment k and execute the nested if statement 300 times
14          {
15                  if (k%100 == 0) //for every 100th print k
16                          cout << k << endl;
17          }
18  }
```

Output



```
100
200
300
```

Loop structure

Our starting condition, ‘k’, is initialised to ‘1’ on line 12. This is followed by the actual ‘`while`’ structure on line 13. The ‘`while`’ loop starts with:

```
while(k++ <= 300)
```

The starting condition is incremented during each loop cycle (via the ‘`k++`’ statement). This condition is true until ‘`k`’ has a value greater than ‘300’; at which point the loop will terminate.

Contained within the while loop is the following ‘`if`’ statement:

```
if (k%100 == 0)
```

This statement prints every 100th value of ‘`k`’.

2.7 The do-while loop

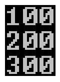
The ‘`do-while`’ loop is another repetition structure used to repeat a sequence of statements or actions. This sequence of statements is repeated infinitely, or until the end condition becomes false. Thus, a ‘`do-while`’ loop will always execute at least once, regardless of whether our test condition is true or not.

The following example is nearly identical to the previous one, only here the ‘`while`’ structure has been modified into a ‘`do-while`’ repetition structure.

Program 2.5 – An example using do-while

```
/*
=================
DoWhile.cpp
- A program using the do-while statement, also features the use of the 'mod' operator
=================
*/

#include<iostream>

using namespace std;

int main()
{
	int k, i; //two counters

	k = 1; //initialise k to 1

	do
	{
		if (k%100 == 0) //for every 100th print k
			cout << k << endl;
	}
	while(k++ <= 300); //increment k and execute the nested if statement 10 times
}
```

Output



```
100
200
300
```

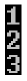
2.8 The for loop

You can use ‘`for`’ loops if you want to evaluate a sequence of expressions for a specified number of times. A ‘`for`’ loop consists of three elements, a control variable, a continuation condition and a control variable counter. The control variable is incremented and compared to the continuation condition during each loop iteration. The loop terminates when the continuation condition is satisfied.

Program 2.6 – An example using the for loop

```
1   /*
2   =================
3   For.cpp
4   - A program using the for repetition structure
5   =================
6   */

7   #include<iostream>

8   using namespace std;

9   int main()
10  {
11          for (int i = 1; i <= 3; i++)
12          {
13                  cout << i << endl;
14          }
15  }
```

Output



```
1
2
3
```

Initialisation, checking and incrementing

The ‘`for`’ loop on line 11 initialises the control variable counter to ‘1’. The loop executes as long as ‘`i`’ is less than or equal to ‘3’ with ‘`i`' incremented by ‘1’ during each iteration.

This chapter concentrated on C++ control structures. The next chapter deals with functions – the very core of the modularity concept (building large programs from smaller units).

Chapter 3

FUNCTIONS

3.1 Preview

You already know something about functions; for instance, Program 1.1 introduced the ‘`main`’ C/C++ entry function. Some functions call themselves *recursively* (over and over) to solve problems that require the continuous evaluation of results. The game at the end of Chapter 6 makes use of function recursion for input control. This chapter will investigate both recursive functions as well as functions that are called once, either from the ‘`main`’ function or from another function.

3.2 The Function Prototype (declaration)

When you declare a function, you’ll need to create a *prototype* for it. This prototype consists of a return type (float, int, void, etc) and the function parameters (variables that will be passed to the function).

Here is an example of what a function prototype could look like:

```
void FunctionOne(string str)
```

The given prototype’s ‘`void`’ keyword indicates that the function doesn’t return anything. ‘`FunctionOne`’ is the function name used to call the function. This can be done either from within the function itself (recursion) or from another function such as the main function. The function name is followed by brackets (‘`()`’) with the function parameters contained within. These function parameters are passed from somewhere else in the program.

The function prototype doesn’t have to list the parameter variable name(s); for example:

```
void FunctionOne(string)
```

3.3 The Function Definition

A function definition encapsulates the code necessary for proper operation of a function:

```
void FunctionOne(string &str)
{
     cout << “func 1”;
}
```

This function accepts a reference to a string ‘`str`’, does nothing with it and prints the line func 1 to the screen. This is, of course, quite a pointless function; why have a reference to a string as a parameter in the first place and never use it? Well, it’s just an example illustrating the proper definition of a function.

Now, say you have declared a string, ‘`userText`’, then you’ll be able to call ‘`FunctionOne`’ from anywhere in your program as follows:

```
string userText = “some default string”; //variable string
FunctionOne(userText);  //calls the function with given string
```

Passing by value occurs when you call a function directly with a variable as a parameter. A local copy of that variable is made and the original cannot be changed. *Passing by reference* can be accomplished through the use of either pointers or references. The function can now manipulate the data contained at the given address, thus changing the value of the original variable. The example demonstrates both options.

You will usually send a couple of values to the function for it to perform some calculation or task, finally returning the result. For example, when the result is an integer you will pre-empt the function name with the ‘`int`’ keyword. Have a look at Table 1.1 for the most common types available.

Example of a function with a return type:

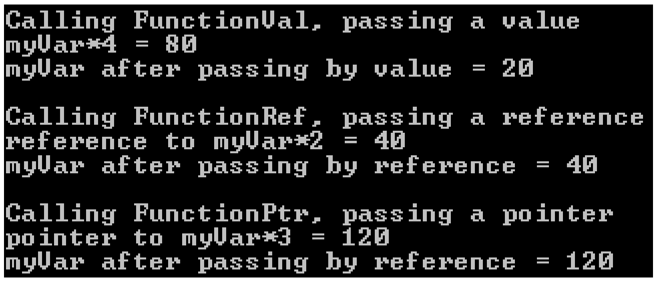
```
int FunctionNumber(int number)
{
     number = number + 2;
     return number;
}
```

Program 3.1 shows a couple of functions in action. The program looks at the direct passing of values, the passing of values by reference and the passing of values via pointers.

Program 3.1 – Functions:

```
/*
=================
Functions.cpp
- A program that illustrates the use of functions
=================
*/

#include<iostream>

using namespace std;

/*
=================
FUNCTION PROTOTYPES
=================
*/
void FunctionRef(short &);
void FunctionPtr(short *);
void FunctionVal(short );

/*
=================
FunctionRef function
- takes a reference as parameter
=================
*/
void FunctionRef(short &someVarRef)
{
        cout << endl << "Calling FunctionRef, passing a reference" << endl;
        someVarRef *= 2;
        cout << "reference to myVar*2 = " << someVarRef << endl;
}

/*
=================
FunctionPtr function
- takes a parameter passed as pointer
=================
*/
void FunctionPtr(short *someVarPtr)
{
        cout << endl << "Calling FunctionPtr, passing a pointer" << endl;
        *someVarPtr *= 3;
        cout << "pointer to myVar*3 = " << *someVarPtr << endl;
}

/*
=================
FunctionVal function
- takes a parameter passed by value
=================
*/
void FunctionVal(short someVar)
{
        cout << endl << "Calling FunctionVal, passing a value" << endl;
        someVar *= 4;
53          cout << "myVar*4 = " << someVar << endl;
54  }

55  /*
56  =================
57  main function
58  - main program
59  =================
60  */
61  int main()
62  {
63          short myVar = 20;
64
65          FunctionVal(myVar); //passing by value
66          cout << "myVar after passing by value = " << myVar << endl;

67          FunctionRef(myVar); //passing by reference
68          cout << "myVar after passing by reference = " << myVar << endl;

69          FunctionPtr(&myVar); //passing by reference using a pointer
70          cout << "myVar after passing by reference = " << myVar << endl;

71          return 0;
72  }
```

Output



```
Calling FunctionVal, passing a value
myVar*4 = 80
myVar after passing by value = 20

Calling FunctionRef, passing a reference
reference to myVar*2  = 40
myVar after passing by reference  = 40

Calling FunctionPtr, passing a pointer
pointer to myVar*3  = 120
myVar after passing by reference  = 120
```

Passing by Value

Have a look at the function, ‘`FunctionVal`’, on line 49. It is clear, from its definition,

`void FunctionVal(short someVar)`

that only one parameter is received. This parameter, ‘`someVar`’, is passed by value.

‘`FunctionVal`’ multiplies the received parameter by ‘4’. The ‘`short`’ integer, ‘`myVar`’, is declared and initialised to ‘20’ on line 63 of the main function. I call ‘`FunctionVal`’ on line 65 using this ‘`myVar`’ variable as follows:

```
FunctionVal(myVar);
```

A local copy of the ‘`myVar`’ variable is made within ‘`FunctionVal`’; this copy is called ‘`someVar`’. ‘`someVar`’ is local to ‘`FunctionVal`’ (no changes made to it are reflected back to ‘`myVar`’, as can be seen from the output).

Passing by Reference

Look at ‘`FunctionRef`’ on line 12. You will notice, upon examination of its definition:

```
void FunctionRef(short &someVarRef)
```

that it receives one parameter passed by reference (specified via the reference operator (‘&’) in front of the parameter ‘`someVarRef`’. This function is invoked via the function call:

```
FunctionRef(myVar);
```

on line 67.

‘`FunctionRef`’ multiplies the reference parameter by ‘2’. Because I am dealing with references, I indirectly multiply variable ‘`myVar`’ in the main program by ‘2’. The variable ‘`myVar`’, initially set to ‘20’, is permanently changed to ‘40’ after execution of ‘`FunctionRef`’. I call (invoke) ‘`FunctionRef`’ on line 67 with the ‘`myVar`’ variable:

```
FunctionRef(myVar);
```

This is where the use of references really becomes apparent. Look at line 69 where I pass a reference using a pointer. With pointers you have to pass the address of the variable, ‘`FunctionPtr(&myVar)`’, making the syntax a bit more complicated. Furthermore, by looking at ‘`FunctionPtr`’ on line 37, you can see the need to dereference the pointer before it can actually be used. The end results from using a reference or pointer is identical; using references makes your code just a little bit easier to read, perhaps saving you hours of debugging time later.

3.4 Function Recursion

I previously touched on the topic of recursion. Recursion is an extremely useful programming tool and as such warrants full coverage. A *recursive function* is a function that calls itself, or that is called directly or indirectly by another function.

To solve a certain problem, the function is written in such a way as to call itself continuously, every time with a simpler version of the original problem. This situation continues until the basis case is encountered. The *basis case* is the most accurate solution to the given problem.

A general issue with recursion is speed. This is mainly due to the same instruction being executed over and over again with only a slight reduction to the problem – more and more memory and time are consumed due to a copy of the function being made during each function call. The most likely situation where you'll implement recursion is where the result of a previous computation is needed for the next.

NOTE: Recursion can take place indefinitely. This happens when the problem isn't reduced in such a manner that it will lead to the basis case.

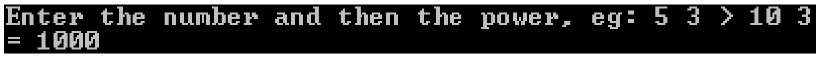
The best way to understand recursion is to see it in action. Let's write a function utilising recursion. Program 3.2 has a function called 'Power'. This function takes two integers, 'p' and 'q', as input parameters and returns the result of 'p' to the power of 'q' ('p' and 'q' are passed by value). The user inputs the numbers 'p' and 'q' and the program prints the result. The program continues until the user enters a power of '0'.

Program 3.2 – Recursive Function

```
/*
=================
RecursivePower.cpp
- A program illustrating the use of recursion
=================
*/

#include<iostream>

/*
=================
FUNCTION PROTOTYPE
=================
*/
int Power(int, int, int);

/*
=================
Power function
- takes three parameters passed by value
@ answ: is the answer of the calculation
@ p: is the base of which we calculate the power
@ q: is the power
=================
*/
int Power(int answ, int p, int q)
```

```
26  {
27          if (q != 0) //base case q = 0
28                  return Power(answ * p, p, q - 1); //recursive call
29          else cout << "= " << answ << endl; //prints the result when base case is met
30  }

31  /*
32  =================
33  main function
34  - main program
35  =================
36  */
37  int main()
38  {
39          int base, power, answ=1;

40          cout << "Enter the number and then the power, eg: 5 3 > ";
41          cin >> base >> power; //reads the user input into the variables
42          Power(answ, base, power); //calls the function Power for the first time

43          return 0;
44  }
```

Output



```
Enter the number and then the power, eg: 5 3 > 10 3
= 1000
```

More than one Parameter

The function, ‘`Power`’ (line 25), receives three parameters (explained in the comments given on lines 19 to 21). The ‘`Power`’ function has a base case of ‘0’; the function is thus called over and over until the base case, q, equals ‘0’. Say the user entered ‘5’ to the power of ‘3’, then the following arithmetic will be performed:

First Call: 1 * 5 ► q – 1 = 3 – 1= **2**
Second Call: 5 * 5 ► q – 1 = 2 – 1= **1**
Third Call: 25 * 5 ► q – 1 = 1 – 1= **0** ► Base Case
Fourth Call: q = 0 thus print the result of ‘`answ`’ = 125

3.5 Function Overloading (Polymorphism)

In short, function overloading or polymorphism refers to the situation where you have two functions with exactly the same name but different parameters. This term polymorphism is frequently used to describe an object that adapts itself to the nature of other objects.

Consider the following function prototypes, signifying three overloaded functions:

```
int OverloadedFunc(int)
int OverloadedFunc(int, int)
int OverloadedFunc(int, long)
```

‘`OverloadedFunc`’ is overloaded three times (notice the differing function parameters which result in different function signatures). The third ‘`OverloadedFunc`’ will be called, for example, when ‘`OverloadedFunc`’ is invoked with ‘`int`’ and ‘`long`’ as parameter types.

NOTE: You cannot have two functions with identical names and parameter types but with different return types.

The idea behind overloading is to make functions more *user-friendly*. User friendliness translates into functions that are easier to use and read. This frees the programmer from worrying about which function to call when dealing with varying variables. If you didn’t use overloading you would have had to create three separate functions for the previous example.

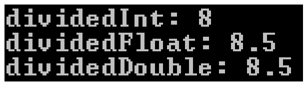
Program 3.3 demonstrates the use of polymorphic functions.

Program 3.3 – Polymorphic Functions

```
/*
=================
Overloader.cpp
- A program illustrating the use of polymorphism
=================
*/

#include<iostream>

/*
=================
FUNCTION PROTOTYPES
=================
*/
int Divide(int);
float Divide(float);
double Divide(double);

/*
=================
int Divide function
- takes 1 integer parameter passed by value and returns integer value
```

```
@ toDivide: the number to be divided
=================
*/
int Divide(int toDivide)
{
            return toDivide/2;
}

/*
=================
float Divide function
- takes 1 float parameter passed by value and returns float value
@ toDivide: the number to be divided
=================
*/
float Divide(float toDivide)
{
            return toDivide/2;
}

/*
=================
double Divide function
- takes 1 double parameter passed by value and returns double value
@ toDivide: the number to be divided
=================
*/
double Divide(double toDivide)
{
            return toDivide/2;
}

/*
=================
main function
- main program
=================
*/
int main()
{
            int intNumber = 17;
            float floatNumber = 17;
            double doubleNumber = 17;

            int dividedInt;
            float dividedFloat;
            double dividedDouble;

            //function calls
            dividedInt = Divide(intNumber);
            dividedFloat = Divide(floatNumber);
            dividedDouble = Divide(doubleNumber);

            //print the results
            cout << "dividedInt: " << dividedInt << endl;
            cout << "dividedFloat: " << dividedFloat << endl;
            cout << "dividedDouble: " << dividedDouble << endl;
73          return 0;
74  }
```

Output



```
dividedInt: 8
dividedFloat: 8.5
dividedDouble: 8.5
```

Overloading in action

The given program overloads the ‘`Divide`’ function for ‘`integer`’, ‘`float`’ and ‘`double`’ parameters, respectively. The different prototypes are given on lines 14 to 16 with the corresponding definitions on lines 25 to 50.

The three ‘`Divide`’ functions have the purpose, that is to divide the passed parameter by ‘2’. The ‘`float`’ and ‘`double`’ ‘`Divide`’ functions return a real number with the ‘`integer`’ version returning the chopped off value, specifically ‘8’ instead of ‘8.5’ (the ‘`integer`’ type cannot store real values).

The next chapter is all about classes – this is also where I will look at the implementation of polymorphism in much more detail.

Chapter 4
CLASSES: OBJECT-BASED PROGRAMMING

4.1 Preview

Classes and objects are pivotal to object-orientated programming. Object-orientation is used to design programs via the definition of objects, inter-object relationships and object-centric properties. It is a widely embraced programming paradigm. Developing programs via object-orientation (known as object-orientated programming (OOP)) allows for easy maintenance and the trouble free implementation of complex methods and types – not always clearly understood nor implemented when using a purely procedural language such as C. Object-orientated programming increases software productivity, reusability and the quality of the software produced. This chapter investigates object-based programming. Object-based programming is a subset of object-orientated programming in that it doesn't make use of inheritance. *Object-based* programming features classes, objects, encapsulation and operator overloading. The *Object-orientated* paradigm adds inheritance and polymorphism to the elements supported by object-based programming. Polymorphism links up with the programming paradigm of generics. *Generic programming*, as it is called, introduces function templates and class templates.

4.2 Classes and Objects

Classes are probably one of the most important characteristics of C++. The class structure is what gives C++ its object-based/orientated quality. Without it there really wouldn't be any real reason to use C++ over C. The class data structure gives the C++ programmer the ability to create custom types. You are, of course, already familiar with the common C++ types such as '`int`', '`float`', '`string`', etc. Then why, you might ask, do we need more data types? The answer lies with the need to solve real world problems. Let's say you want to create a fight simulator, then you can use classes to define custom variables for the representation of gauges, throttle control, landing gear status, etc. These new types, defined by classes, are called ADTs or *Abstract Data Types*. Classes facilitate the creation of objects consisting of attributes and operations. *Attributes* of an object are represented as data members. *Operations* on the other hand are represented as member functions. These member functions are similar to the functions discussed in Chapter 3. Member functions are invoked by messages sent to objects. Data members are nothing more than variables used in the definition of member functions. All these members are guarded via a form of access control.

Another concept associated with object-based/orientated programming is encapsulation. *Encapsulation*, or information hiding, allows one to specify either '`public`', '`private`' or '`protected`' visibility for the members of a class. *Public members* are accessible from anywhere within the program. *Private members* are only accessible from the same class's member functions or from other classes and their member functions explicitly declared as friends of the class. *Protected members* are accessible by all derived classes (classes inheriting the base, or original, class); protected members are also accessible by friends of the class as well as the class itself.

NOTE: A class definition terminates with a semicolon. Failure to do so will result in a syntax error.

You will now look at a complete object-based C++ program.

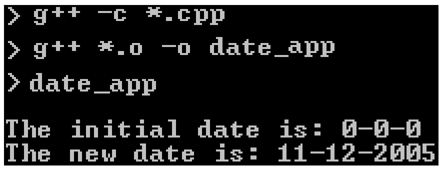
Program 4.1 – The object-based programming

```
/*
=================
Date.h
- Header file for class definition - SPECIFICATION
=================
*/

#ifndef DATE_H
#define DATE_H

#include <iostream> //no need to include it anywhere else where Date.h is included

using namespace std;

class Date
{
        public:                         //public access modifier
                Date();
                void setDate(int, int, int);
                void printDate();
        private:                //private access modifier
                int year;
                int month;
                int day;
};

#endif
```

```
/*
=================
Date.cpp
- Header file for class definition - IMPLEMENTATION
=================
*/

#include "Date.h"

/*
=================
- Data constructor initializes each data member to 0
- Is always called, thus ensures all Date objects are in a consistent state
=================
*/
Date::Date()
{
		day = 0;
		month = 0;
		year = 0;
}

/*
=================
- Sets new Date value, note: no validity checking is done
=================
*/
void Date::setDate(int day_, int month_, int year_)
{
		day = day_;
		month = month_;
		year = year_;
}

/*
=================
- Prints the date in international format
=================
*/
void Date::printDate()
{
		cout << year << "-" << month << "-" << day << endl;
}


/*
=================
main.cpp
Main entry point for the Date application
=================
*/

#include "Date.h"

int main()
{
		Date d;		//instantiate object d of class Date
72          cout << "The initial date is: ";
73          d.printDate(); //calls the values as initialised by the default constructor

74          d.setDate(2005, 12, 11);
75          cout << "The new date is: ";
76          d.printDate();

77          return 0;
78  }
```

NOTE: The specification and implementation of C++ programs are separated. Program 4.1 consists of three physical files: Data.h (the specification), Data.cpp (the implementation) and main.cpp (the application's entry point).

Compilation and Output



```
> g++ -c *.cpp
> g++ *.o -o date_app
> date_app

The initial date is: 0-0-0
The new date is: 11-12-2005
```

Definition

The '`Date`' class definition starts with the '`class`' keyword. You contain a set of statements within the body of the class, specified by left ('`{`') and right ('`}`') braces. A class definition is terminated with a right brace followed by a semicolon ('`;`').

The conditional compilation directives: '`#ifndef`', '`#define`' and '`#endif`' are used on lines 7, 8 and 22 independently. In the example program's context, this directive grouping ensures that if '`Date.h`' has already been included, then it will not be included again, otherwise it will be included (thus preventing multiple inclusions of the '`Date.h`' header file). You'll generally use conditional compilation directives to facilitate cross-platform compatibility.

The '`Data`' class definition contains three '`private`' integer members: '`year`', '`month`' and '`day`'. The default class access specifier is '`private`', so you could have written the class definition with the integer members at the top of the class definition (omitting the '`private`' access modifier – which is default unless you specify something else, e.g. '`public`' or '`protected`'):

```
class Date
{
     //members are private by default
     int year;
     int month;
     int day;

     public:   //public access modifier
          Date();
          void setDate(int, int, int);
          void printDate();
};
```

Member Prototypes

Three member functions are defined on lines 14, 15 and 16 (all publicly visible). The first member function definition is for the default constructor of the class,

```
Date();
```

The default constructor is a constructor without any arguments.

NOTE: The default constructor is generated by the compiler if none has been specified.

A *constructor* is a type of member function, more specifically; it is a member function with the same name as the class. A constructor is invoked each time the program creates an object of the class.

On line 36 a constructor is used to initialise each data member to '0'.

NOTE: A constructor isn't assigned a return type and there is no restriction on its functionality.

The declaration of our '`setDate`' and '`printDate`' member functions,

```
void setDate(int, int, int);
printDate();
```

on lines 15 and 16, are used by the clients of this class to manipulate the class data, in essence the integer members ‘`year`’, ‘`month`’ and ‘`day`’. These member functions allow the client’s code to interact with the objects of the class.

Using the class

Let’s look at the usage of this ‘`Date`’ class. A single object, ‘d’, is instantiated on line 71. This object is of type ‘`Date`’. The ‘`Date`’ constructor is called upon object instantiation, leading to the initialisation of each of the private data members ‘`year`’, ‘`month`’ and ‘`day`’.

In the class implementation, ‘`Date.cpp`’, notice our usage of the scope resolution operator (‘`::`’). It is used to refer to a class member function rather than a local one. In practical terms this just means that if you write ‘`classname::function()`’, then you are referring to a member function located in the declaration of ‘`classname`’. In our program, on lines 47 to 52, I define the member function ‘`setDate`’ externally via this scope resolution operator.

```
void Date::setDate(int day_, int month_, int year_)
{
     day = day_;
     month = month_;
     year = year_;
}
```

There isn’t really that much more to say about this little program, except perhaps for the code on lines 73 to 76:

```
d.printDate();
```

prints the data members as initialised by the constructor.

NOTE: You can also use pointers and references to access member functions. This is discussed and illustrated in section 4.3.

The following section of code:

```
d.setDate(2001, 12, 11);
cout << "The new date is: ";
d.printDate();
```

calls the member function ‘setDate’ with a date (year, month, day) as parameter, printing the changed output via the ‘printdate’ member function.

NOTE: Using information hiding techniques, such as those promoted by access modifiers, improve program modifiability and readability.

It is also important to note that function variables (variables defined in a member function) are only visible in the containing function – no other function or class can gain access to them, ever. For example, say you add a function variable, ‘int x’, to the ‘printDate’ member function:

```
void Date::printDate()
{
     int x;
     x = 7;
     cout << year+x<< "-" << month << "-" << day << endl;
}
```

then ‘x’ will only be visible and accessible by the ‘printDate’ function.

The ‘printDate’ function will now print the year value to the screen, but added to it the value 7. You can pretty much do anything to this function variable as long as it is within the ‘printDate’ member.

4.3 Access of Class Members

Every class has a specific scope. This scope is defined by data members (class variables) and member functions (those functions unique to a class). When working within the scope of a class, class members are accessed via the class’s member functions. When working from outside the class’s scope, you can reference the class members via an object name, a pointer to an object or a reference to an object.

Accessing a class object via the object’s name, a reference to the object or via a pointer is best illustrated in the example below. These three different access techniques are known as *object handles*. The *dot member selection operator* (‘.’) is used in conjunction with the object name. This selection operator can also be combined with an object reference to access the object’s members. The alternative *arrow member selection operator* (‘->’) is used in combination with a pointer to the object.

So why use different methods if the end result is always the same? The answer lies with the fact that class instances can either be *heap dynamic* or *stack dynamic*. Class instances are referenced through pointers when heap dynamic and directly via value if stack dynamic. This use becomes clear when looking at the storage bindings of variables. As you know by now, for a variable to exist it must be bound to a memory cell, this memory cell is acquired from the computer's available physical memory. This is the process of *allocation*. *Deallocation* is the exact reverse process – freeing the bounded memory cells and thus making it available for future use. The *heap* and *stack* are two different memory areas (in addition to the global name space, registers and the code space memory areas).

The *code space* is used for code with the *global namespace* used for all our global variables. The *stack* is primarily used for function parameters and local variables and its associated *registers* are used for the internal organisation of the stack. The *heap* is very important because of nearly all our free memory being allocated to it. Thus, by using the different object handles, you can control the memory binding process.

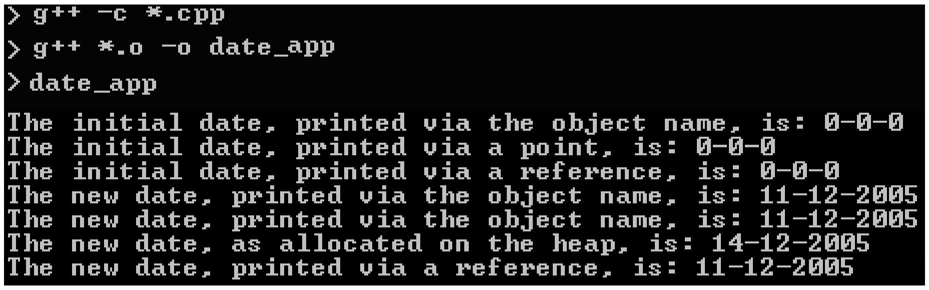
Heap memory allocation is advantageous due to its persistency. For example, when a member function terminates, its associated variables are not destroyed; as the case with stack based allocation. Once you've reserved memory on the heap, it remains reserved until explicitly freed. But, just as it must be explicitly freed, it must also be explicitly allocated. You allocate memory on the heap via the '`new`' keyword. This allocated memory is in turn freed via the '`delete`' keyword. These two keywords aren't available in the C programming language, however, you can make use of '`malloc`' and '`free`' for heap allocation and deallocation, respectively.

Program 4.2 – Modified version of Program 4.1 illustrating object handles and heap allocation

```
/*
=================
Date.h
- Header file for class definition - SPECIFICATION
=================
*/

#ifndef DATE_H
#define DATE_H

#include <iostream> //no need to include it anywhere else where Date.h is included

using namespace std;

class Date
{
        public:             //public access modifier
                Date();
                void setDate(int, int, int);
                void printDate();
```

```
		private:		//private access modifier
			int year;
			int month;
			int day;
};

#endif

/*
=================
Date.cpp
- Header file for class definition - IMPLEMENTATION
=================
*/

#include "Date.h"

/*
=================
- Data constructor initializes each data member to 0
- Is always called, thus ensures all Date objects are in a consistent state
=================
*/
Date::Date()
{
		day = 0;
		month = 0;
		year = 0;
}

/*
=================
- Sets new Date value, note: no validity checking is done
=================
*/
void Date::setDate(int day_, int month_, int year_)
{
		day = day_;
		month = month_;
		year = year_;
}

/*
=================
- Prints the date in international format
=================
*/
void Date::printDate()
{
		cout << year << "-" << month << "-" << day << endl;
}

/*
=================
main.cpp
Main entry point for the Date application
=================
*/

#include "Date.h"

int main()
{
		Date d;		//create the Date object d
		Date *dPtr = &d; //create the pointer, dPtr to the Date object d
		Date &dRef = d; //create the reference, dRef to the Date object d

		Date *heapDate = new Date(); //heap allocation

		cout << "The initial date, printed via the object name, is: ";
		d.printDate(); //calls the values as initialised by the default constructor

		cout << "The initial date, printed via a point, is: ";
		dPtr->printDate(); //calls the values as initialised by the default constructor

		cout << "The initial date, printed via a reference, is: ";
		dRef.printDate(); //calls the values as initialised by the default constructor

		d.setDate(2005, 12, 11);
		cout << "The new date, printed via the object name, is: ";
		d.printDate();

		dPtr->setDate(2005, 12, 11);
		cout << "The new date, printed via the object name, is: ";
		dPtr->printDate();

		heapDate->setDate(2005, 12, 14);
		cout << "The new date, as allocated on the heap, is: ";
		heapDate->printDate();

		dRef.setDate(2005, 12, 11);
		cout << "The new date, printed via a reference, is: ";
		dRef.printDate();

		delete heapDate; //free memory

		return 0;
}
```

Compilation and Output



```
> g++ -c *.cpp
> g++ *.o -o date_app
> date_app
The initial date, printed via the object name, is: 0-0-0
The initial date, printed via a point, is: 0-0-0
The initial date, printed via a reference, is: 0-0-0
The new date, printed via the object name, is: 11-12-2005
The new date, printed via the object name, is: 11-12-2005
The new date, as allocated on the heap, is: 14-12-2005
The new date, printed via a reference, is: 11-12-2005
```

Accessing class members

Program 4.2 is a near identical duplication of Program 4.1. Even the output, discounting Program 4.2's numerous printouts, is identical. However, Program 4.2 features two additional object handles (access by pointers and references). It also implements heap allocation via use of the 'new' operator.

NOTE: C++ supports access to an object's data members and member functions through 3 types of object handles (name, pointer and reference).

The change you'll see, when looking at this program's main function, is the creation of four objects instead of Program 4.1's one.

The first created object is the same as Program 4.1's:

```
Date d; //create the Date object d
```

Two additional object handles are added on lines 71 and 72:

```
Date *dPtr = &d; //create the pointer, dPtr to d
Date &dRef = d; //create the reference, dRef d
```

You are encouraged to play around with these variants. The given source code is thoroughly documented and clearly highlights the different calls and initialisations (line 74 to 91). In addition, the given program output shows the end result of using different object handles.

I use the new keyword on line 73. This results in the 'heapDate' object's heap allocation.

```
Date *heapDate = new Date();
```

The 'delete' operator on line 92 is responsible for the 'deallocation' of the 'heapDate' object's memory.

NOTE: Never dereference a pointer after deallocating the object.

The object's destructor is called upon deallocation of the class object. Program 4.2 doesn't have a destructor; hence no destructor will be invoked. I mentioned that a constructor can be considered a member function with the exception of being invoked every time an object is created. A destructor is the exact opposite and is called every time an object is deallocated. You signify a member function as a destructor by preceding the function name with a tilde ('~'). In addition this, a destructor's function name must be the same as the class name (just as the case with a constructor). For example, say you had a class called '`Render`', then you'd create a constructor by specifying a public member function '`Render()`' (that may or may not take any parameters) and a destructor '`~Render()`' which won't have any parameters. This is illustrated below:

```
/*
=================
Define Class
=================
*/
class Render
{
public:
     Render(volume *dimensions); //declare our constructor
     ~Render(); //declare the destructor.
private:
     string *some_literal;
};

/*
=================
Define Constructor
=================
*/
Render::Render(volume *dimensions)
{
     //allocate dynamic memory
     _image = new Image(dimensions);

}

/*
=================
Define Constructor
=================
*/
```

```
Render::~Render()
{
     //deallocate reserved memory
     delete _image;
}
```

So there you have it; everything to get you started with memory allocation and the creation and access of objects. I now proceed on to the last two sections of object based programming, namely encapsulation and operator overloading. Both encapsulation and operator overloading contribute to the paradigm of good software engineering and are greatly used in the game and modern software development industry. Without encapsulation and operator overloading it would be impossible to accomplish the design aims of good readability, modifiability and the vital maintenance of large programs.

4.4 Encapsulation

Section 4.2 outlined the difference between data members (the variables defined within a C++ class) and member functions (the class functions where the actual work is performed). Section 4.3 showed how a single set of member functions is shared by multiple instances of a class and how every instance of the class is assigned its own set of data members. What this chapter will look at now is the concept of information hiding, more commonly referred to as *encapsulation*.

The primary motivation behind encapsulation is that some of the details in the definition of a class might be irrelevant for ordinary clients. Encapsulation is thus the hiding of this irrelevant information.

These irrelevant details often include the representation of the objects of a class, the state required to maintain these objects and the member functions used to respond to messages of a class.

When dealing with encapsulation, you firstly divide a program into a declarative view and an operational view. The *declarative view* is commonly referred to as the specification or interface of the program. The *operational view* is referred to as the implementation or body of the program. The *specification* can, in turn, be defined as a collection of data and method definitions (called signatures) and descriptions (usually comments). The body of the program is the representation of the data and the implementation of the operations.

These are nothing more than a couple of formal definitions but it is these definitions that are partially to blame for all those '`.h`' and '`.cpp`' files you've created so far. Many modern programming languages like Java and C# use an integrated specification and implementation (with specialised tools for the extraction of the specification). In C++, you use '`.h`' and '`.cpp`' files to physically separate the specification and implementation. Thus, just like in the previous examples, you'll define your classes in '`.h`' files and all your member functions in '`.cpp`' files.

The level of separation isn't only limited to classes and functions but the class definition is also separated into a private and public interface. All these separated interfaces and files are, in the end, assembled into executable programs by the pre-processor.

4.4.1 Friends

Having a class definition separated into private and public interfaces can lead to problems when it is accessed from the outside, specifically, when inheritance is used. Now, although inheritance isn't part of object-based programming (falling under object orientation) it forms part of encapsulation by classes allowing other classes or member functions of other classes to be declared as *friends*.

A class, declared as a friend of another class, has access to all the private members of that friend class. Likewise, declaring a member function as a friend of some other class will grant access to all the private members of that friend class.

This is best illustrated by the following two code snippets.

```
class Matrix4x4
{
     friend char Matrix::DotProduct();
…
};
```

Here you have the definition of some class '`Matrix4x4`' containing the code:

```
friend char Matrix::DotProduct();
```

This code portion shows that member function '`DotProduct`', inherited from class '`Matrix`', is declared a friend of class '`Matrix4x4`'. Inheritance is indicated with a double colon, also called the scope operator ('`::`'); which, in this case, tells the compiler to interpret '`DotProduct`' within the context of '`Matrix`'. Hence, a class gains access to the members of some other class via inheritance. The member function, '`DotProduct`', will thus have access to all private members of the friend class '`Matrix4x4`'.

Similarly, in the code extract below, the entire class ‘`Render`’ is declared a friend of class ‘`Matrix`’ (giving it access to all the private member functions of class ‘`Matrix`’).

```
class Matrix
{
     friend class Render;
     …
};
```

NOTE: A class definition may consist of any number of private and public interfaces.

4.4.2 Namespaces

Namespaces enhance encapsulation by acting as a scope for the definition of elements. All the previous sample programs included the following code segment:

```
using namespace std;
```

This is a predefined namespace called ‘`std`’. The ‘`std`’ namespace holds a number of predefined classes and predefined functions, for example, the already familiar ‘`cout`’ function. A function must always be qualified with the name of its namespace. For example; to use ‘`cout`’ without including the ‘`using namespace`’ code segment, you will have to call it like this:

```
std::cout << ”Some string…” << std::endl;
```

Just as with this standard namespace, you can also declare namespaces for different parts of a program. This allows you to use the same class names in different namespaces. A game can make use of namespaces to partition its engine into distinct sections by declaring different namespaces for its renderer, physics engine, networking subsystem, etc. Thus, when initialising the renderer at the start of the game, you’ll qualify the class (‘`GLDriver`’ in this case) with the namespace (‘`renderer`’ in this case) as follows:

```
renderer::GLDriver* renderer = Renderer->detectVideoDrv();
```

4.5 Operator Overloading

Operator overloading gives one the power to change the meaning of an operator. For example, the addition (‘+’) operator can be overloaded to represent a cross product.

The ‘`operator`’ keyword is used to facilitate this overloading. Most built-in operators can be redefined on a class-by-class basis. They can also be redefined in a global fashion; however, this is rarely done. After defining the overloaded operator, you implement them as you would normal member functions.

Let’s look at overloading the addition (‘+’) operator. The name of the overloaded operator is ‘`operator+`’. The overloaded operator, in this case `+`, always follows the operator keyword. Table 4.1 gives the complete list of overloadable C++ operators.

Table 4.1 – C++ Overloadable Operators

Operator	Description
+	Binary Addition
-	Binary Subtraction
+	Unary Plus
-	Unary negation
++	Increment
--	Decrement
+=	Addition assignment
==	Equality
!=	Inequality
-=	Subtraction assignment
*	Multiplication
/	Division
*=	Multiplication assignment
/=	Division assignment
%=	Modulus assignment
^=	Exclusive OR assignment
|=	Bitwise inclusive OR assignment
&=	Bitwise AND assignment
<<=	Left shift assignment
>>=	Right shift assignment
&	Bitwise AND
&&	Logical AND
!	Logical NOT
|	Bitwise inclusive OR
^	Exclusive OR
||	Logical OR
<<	Left shift
>>	Right shift
<	Less than
>	Greater than
=	Assignment
%	Modulus
~	One's complement
*	Pointer dereference

&	Memory-Address
[ ]	Array subscript
new	new keyword
delete	delete keyword
->	Member selector
->*	Pointer-to-member selector
( )	Function calling
,	Comma

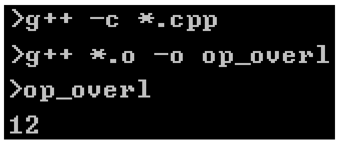
The following program illustrates the use of overloaded operators. It specifically overloads the binary addition operator ('`+`') to return a new '`Float`' object, thus avoiding modification of the object from which the operator is called. This returned '`Float`' object is cast to an '`integer`'. Program 4.3 doesn't have any "real life" application but it clearly illustrates the concept of operator overloading.

Program 4.3 – Overloading Operators

```
/*
=================
Float.h
- Header file for class definition - SPECIFICATION
- The example overloads the + operator to add two float values together,
  casting the result to int
=================
*/

#ifndef FLOAT_H
#define FLOAT_H

#include <iostream>

using namespace std;

class Float
{
public:
   Float(float);
   Float operator+(Float &);
   void Output();
private:
   float value;
};

#endif

/*
=================
Float.cpp
- C++ file for class definition - IMPLEMENTATION
=================
*/

#include "Float.h"
```

```
/*
=================
- Float constructor initializes val data member to 0
- Is always called, thus ensures all Float objects are in a consistent state
=================
*/
Float::Float(float i)
{
             value = i;
}

/*
=================
- overloading the binary addition operator using a member function then casting to int
=================
*/
Float Float::operator+(Float &i)
{
            return Float((int)(value + i.value));
}

/*
=================
- prints the output to the screen
=================
*/
void Float::Output()
{
            cout << value << endl;
}

/*
=================
main.cpp
=================
*/

#include "Float.h"

int main()
{
   Float a = Float(9.5);
   Float b = Float(3.2);
   Float c = Float(0.0);

   c = a + b;

   c.Output();
   return 0;
}
```

Compilation and Output



```
>g++ -c *.cpp
>g++ *.o -o op_overl
>op_overl
12
```

Overloading

The code segment:

```
Float operator+ (Float &);
```

on line 17 defines the binary addition overloaded operation (with a ‘`Float`’ type as specified by the constructor on line 16) for floating-point types. This definition is implemented on line 47. The client uses this overloaded operator as follows (see the main section of the program):

```
Float a = Float(9.5);
Float b = Float(3.2);
Float c = Float(0.0);

c = a + b;
```

The given four lines of code create the ‘`Float`’ objects ‘a’, ‘b’ and ‘c’. These objects are initialised with the values 9.8, 3.2 and 0.0, respectively. The ‘`Float`’ types are then added on line 69:

```
c = a + b;
```

with this operation translated as:

```
c = a.operator+(b);
```

So by defining a new meaning for an existing, built-in operator, you have the power to add your own custom objects together, for example. In addition, operator overloading promotes much more compact and robust code. Operator overloading does, however, come with a couple of guidelines. One of the most important guidelines worth remembering is that you should always use operating overloading in a symmetrical manner, that is, if you overloaded the left shift (‘<<’) operator, then you should also overload the right shift (‘>>’) operator. Another guideline is that you should only use operator overloading in cases where its use is immediately apparent, for example overloading the subtraction operator (‘-’) for the calculation of the distance between two volumetric objects in space makes perfect sense, but using, say, the addition operator as I did in our example program to cast the sum of two float values to integer doesn’t really make any sense in “real life”.

I previously identified inheritance and polymorphism as two features of the object-orientated paradigm, touching on inheritance during the section on friends and dealing with function polymorphism in section 3. I now move on to object-orientated programming, focussing in more detail on inheritance, polymorphism and other aspects of object-orientation.

Chapter 5
CLASSES: OBJECT-ORIENTATED PROGRAMMING

5.1 Preview

Extending pre-defined classes into new ones has become pivotal to modern program design and is the most important feature of object-orientated programming. This extension of classes is called inheritance. There are three kinds of inheritance: public, private and protected. I will focus on public inheritance. The final part of the chapter introduces the concept of polymorphism (using a single definition for different types of data). Two types of polymorphism will be touched on: ad-hoc polymorphism (using a finite range of types) and parametric polymorphism (no finite range of types, thus allowing your code to work with any type of parameters).

5.2 Inheritance

Inheritance, often termed generalisation, is based on an *is a* relationship due to the hierarchical structure between classes and objects. Let's look at an every day example. Say you have a car consisting of several components: a steering wheel, four tires and obviously numerous other components, each quite necessary for the vehicles operation. Now, inheritance provides a way for the creation of new classes from old ones, hence, in the case of our car example, '`car`' is a generalisation of the vehicle's make. So '`car`' is a *generalisation* of BMW or VW. But because BMW *is a* car, it inherits all these common components a car consists of (the steering wheel, the four tires, etc…).

Our trivial example illustrates the importance of inheritance – the complexity of a program can be reduced by avoiding the redefinition of classes and member functions. In essence, one class shares code already defined in another class. The class where the code is shared from is known as the *base class* with the class inheriting this code and attributes as the *derived class*.

5.2.1 UML Diagrams

The Unified Modeling Language (UML) is a standard graphical method for the modeling of object-orientated software. It originated in the mid-90s as a collaboration effort between James Rumbaugh, Grady Booch (a pioneer in object-orientation) and Ivar Jacobson (each of these developers defined their own modeling language in the early 90s). The UML standard is governed by the Object Management Group (OMG).

Visual models of a software system are crucial for an effective software development process. The large software systems common today are extremely difficult to describe and maintain without such visual aids. With the use of UML, everyone can interpret the design of a program in the same way.

In this chapter I use UML diagrams to represent class diagrams (describing classes and their relationships due to inheritance). UML diagrams are also used in the industry for interaction diagrams (shows the behaviour of systems via object interaction), state diagrams (shows internal system behaviour) and component diagrams (shows the logical and physical arrangement of system components).

Figure 5.1 gives the UML class diagram for our car example. The '`Wheel`' class is the base class with the '`BMW`' and '`VW`' classes the derived classes inheriting from the '`Wheel`' class. The book, *3D Game Programming with DirectX 10 and OpenGL*, deals with UML diagrams in much more detail.

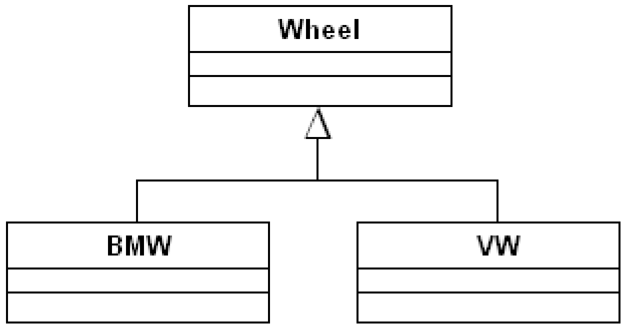


Fig 5.1 UML diagram showing a base class and its derived classes

5.2.2 Public Inheritance

Public inheritance gives a derived class access to the public interface of the base class. This public interface includes all public data members and public member functions. Public inheritance is best illustrated in Program 5.1.

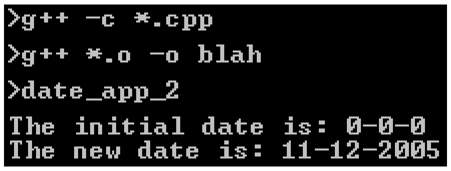
Program 5.1 – Public Inheritance

```
/*
=================
SetDate.h
- Header file for class definition - SPECIFICATION
=================
*/

#ifndef SETDATE_H
#define SETDATE_H

#include <iostream>

using namespace std;

class SetDate
{
        public:                 //public access modifier
                SetDate();
                void setDate(int, int, int);
                void printDate();
        private:                //private access modifier
                int year;
                int month;
                int day;
};

#endif

/*
=================
SetDate.cpp
- Header file for class definition - IMPLEMENTATION
=================
*/

#include "SetDate.h"

/*
=================
- SetDate constructor initializes each data member to 0
- Is always called, thus ensures all Date objects are in a consistent state
=================
*/
SetDate::SetDate()
{
        day = 0;
        month = 0;
        year = 0;
}

/*
=================
- Sets new Date value, note: no validity checking is done
=================
*/
void SetDate::setDate(int day_, int month_, int year_)
```

```
{
            day = day_;
            month = month_;
            year = year_;
}

void SetDate::printDate()
{
            cout << year << "-" << month << "-" << day << endl;
}

/*
=================
Date.h
- Header file for class definition - SPECIFICATION
=================
*/

#ifndef DATE_H
#define DATE_H

#include "SetDate.h"

class Date : public SetDate
{
            public:                             //public access modifier
                        void displayResults();
};

#endif

/*
=================
Date.cpp
- Header file for class definition - IMPLEMENTATION
=================
*/

#include "Date.h"

/*
=================
- Prints the date in international format
=================
*/
void Date::displayResults()
{
            printDate();
}

/*
=================
main.cpp
Main entry point for the Date application
=================
*/

```

```
95   #include "Date.h"

96   int main()
97   {
98           Date d;            //instantiate object d of class Date
99
100          cout << "The initial date is: ";
101          d.displayResults(); //calls the values as initialised by the default constructor

102          d.setDate(2005, 12, 11);
103          cout << "The new date is: ";
104          d.displayResults();

105          return 0;
106  }
```

Compilation and Output



```
>g++ -c *.cpp
>g++ *.o -o blah
>date_app_2
The initial date is: 0-0-0
The new date is: 11-12-2005
```

Inheritance

The given program is based on Program 4.1. Program 4.1 has been modified in such a way as to give the derived class (‘`Date`’) access to the entire public interface of the base class (‘`SetDate`’). The key code snippet, making all of this possible, is located on line 66:

```
class Date : public SetDate
```

The declared class ‘`Date`’ is thus derived from the class ‘`SetDate`’.

The ‘`SetDate`’ class is identical to Program 4.1’s ‘`Date`’ class. Only the public interfaces are inherited from ‘`SetDate`’. This gives ‘Date’ access to the member functions of ‘`SetDate`’. The member functions ‘`setDate`’ and ‘`printDate`’ are thus directly accessible via the ‘`Date`’ object ‘d’ created in the main section of the program (line 98). The main functions of Program 4.1 and 5.1 are identical except for line 104 where I call:

```
d.displayResults();
```

instead of

```
d.printDate();
```

The private attributes (year, month, day) declared in ‘`SetDate`’ are not accessible from anywhere outside the ‘`SetDate`‘ class. Therefore, calling ‘`printDate`’ on line 104 will result in a compiler error. The ‘`displayResults`’ member function is thus declared to avoid this.

Inheritance allows you to call the member function ‘`printDate`’ from within the ‘`Date`’ class as you would a local function:

```
void Date::displayResults()
{
     printDate();
}
```

5.2.3 A Mention of Private & Protected Inheritance

Section 4.2 illustrated that any data member or member function declared as private can only be accessed by the member functions and friends of that class. Additionally, it noted that any data member or member function declared as protected is only accessibly by member functions and friends of that class, including the member functions and friends of derived classes. Data members or member functions declared as public are globally accessible.

With *private inheritance*, all the base data members and member functions are treated as private regardless of their actual declaration. More simply stated and based on a public inheritance example of a *BMW being a CAR* (hence CAR being the base class and BMW the derived class); with private inheritance, BMW is not a CAR. BMW is rather implemented in terms of CAR.

Private inheritance is thus used when you'd like to implement a new object in terms of an existing object and when an *is a* relationship is inappropriate.

To indicate private inheritance, use of the private keyword as follows:

```
class Date : private SetDate
```

The only difference between private and *protected inheritance* is that protected members of the base class are treated as protected in the derived class as opposed to private (as the case with private inheritance). Protected inheritance is indicated with the protected keyword preceding the base class name in the class designation:

```
class Date : protected SetDate
```

5.2.4 Multiple Inheritance

C++ has a feature that allows derived classes to inherit from multiple base classes. *Multiple inheritance*, as this feature is called, has been removed from many post-C++ languages such as Java and C# (although there are ways around this in C#) because, although useful at times, multiple inheritance can create a structural mess when misused.

Inheritance simplifies a program because it eliminates the need for multiple definitions of the same thing. Multiple inheritance furthers this idea. The classes inherited from can be unrelated but needn't be.

Practically, to indicate multiple inheritance; you separate the base classes with commas (‘`,`’). For example, the code snippet below defines the class ‘`Date`’ to inherit from both ‘`SetDate`’ and ‘`SetTime`’:

```
class Date : public SetDate, public SetTime
```

NOTE: The base class initialises the constructors of all the derived classes.

In Figure 5.2, the ‘`Wheel`’ and ‘`RevCounter`’ classes are the base classes with the ‘`BMW`’ class the derived class.

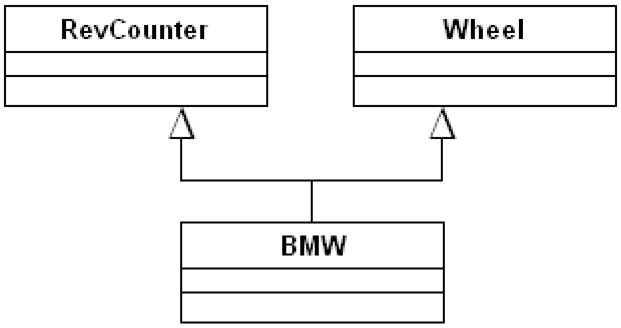


Fig 5.2 UML diagram showing multiple inheritance.

Program 5.2 – Multiple Public Inheritance

```
/*
=================
SetDate.h
- Header file for class definition - SPECIFICATION
```

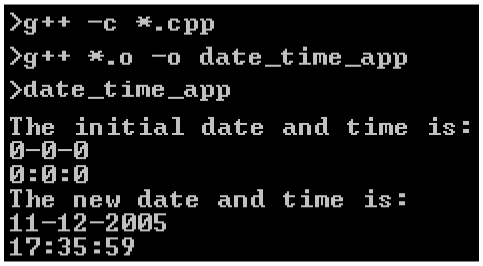
```
=================
*/

#ifndef SETDATE_H
#define SETDATE_H

#include <iostream>

using namespace std;

class SetDate
{
        public:                     //public access modifier
                SetDate();
                void setDate(int, int, int);
                void printDate();
        private:                    //private access modifier
                int year;
                int month;
                int day;
};

#endif

/*
=================
SetDate.cpp
- Header file for class definition - IMPLEMENTATION
=================
*/

#include "SetDate.h"

/*
=================
- SetDate constructor initializes each data member to 0
- Is always called, thus ensures all Date objects are in a consistent state
=================
*/
SetDate::SetDate()
{
        day = 0;
        month = 0;
        year = 0;
}

/*
=================
- Sets new Date value, note: no validity checking is done
=================
*/
void SetDate::setDate(int day_, int month_, int year_)
{
        day = day_;
        month = month_;
        year = year_;
}
```

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

/*
=================
SetTime.h
- Header file for class definition - SPECIFICATION
=================
*/

#ifndef SETTIME_H
#define SETTIME_H

#include <iostream>

using namespace std;

class SetTime
{
        public:                 //public access modifier
                SetTime();
                void setTime(int, int, int);
                void printTime();
        private:        //private access modifier
                int minute;
                int hour;
                int second;
};

#endif

/*
=================
SetTime.cpp
- Header file for class definition - IMPLEMENTATION
=================
*/

#include "SetTime.h"

/*
=================
- SetTime constructor initializes each data member to 0
- Is always called, thus ensures all Time objects are in a consistent state
=================
*/
SetTime::SetTime()
{
        hour = 0;
        minute = 0;
        second = 0;
}

/*
```

```
=================
- Sets new Time value, note: no validity checking is done
=================
*/
void SetTime::setTime(int hour_, int minute_, int second_)
{
		hour = hour_;
		minute = minute_;
		second = second_;
}

void SetTime::printTime()
{
		cout << hour << ":" << minute << ":" << second << endl;
}

/*
=================
Date.h
- Header file for class definition - SPECIFICATION
=================
*/

#ifndef DATE_H
#define DATE_H

#include "SetDate.h"
#include "SetTime.h"

class Date : public SetDate, public SetTime
{
		public:				//public access modifier
			void displayResults();
};

#endif

/*
=================
Date.cpp
- Header file for class definition - IMPLEMENTATION
=================
*/

#include "Date.h"

/*
=================
- Prints the date in international format
=================
*/
void Date::displayResults()
{
		printDate();
		printTime();
}
```

```
147  /*
148  =================
149  main.cpp
150  Main entry point for the Date application
151  =================
152  */

153  #include "Date.h"

154  int main()
155  {
156          Date d;          //instantiate object d of class Date

157          cout << "The initial date and time is: ";
158          d.displayResults(); //calls the values as initialised by the default constructor

159          d.setDate(2005, 12, 11);
160          d.setTime(17, 35, 59);
161          cout << "The new date and time is: ";
162          d.displayResults();

163          return 0;
164  }
```

Compilation and Output



```
>g++ -c *.cpp
>g++ *.o -o date_time_app
>date_time_app
The initial date and time is:
0-0-0
0:0:0
The new date and time is:
11-12-2005
17:35:59
```

Multiple Inheritance

Program 5.1 has been modified to include a class, ‘`SetTime`’, for the setting and printing of the time as specified by the object (line 160). The derived class, ‘`Date`’, has access to the entire public interface of the base classes ‘`SetDate`’ and ‘`SetTime`’. Multiple inheritance is specified on line 124:

```
class Date : public SetDate, public SetTime
```

Thus, class ‘`Date`’ is derived from the classes ‘`SetDate`’ and ‘`SetTime`’.

Just as with Program 5.1, the ‘`SetDate`’ class is identical to the ‘`Date`’ class of Program 4.1 and only the public interfaces are inherited from ‘`SetDate`’ (and now additionally ‘`SetTime`’). ‘`Date`’ therefore has access to the member functions of ‘`SetDate`’ and

‘SetTime’. The member functions ‘setDate’, ‘setTime’, ‘printDate’ and ‘printTime’ are in effect directly accessibly by the ‘Date’ object ‘d’ created in the main section of the program (line 156). The main function of Program 5.1 has been altered to include the initialisation of the ‘SetTime’ constructor:

```
d.setTime(17, 35, 59);
```

Because the program makes use of multiple inheritance; I call the member functions ‘printDate’ and ‘PrintTime’ in the ‘Date’ class as I would a local function:

```
void Date::displayResults()
{
     printDate();
     printTime();
}
```

5.3 Polymorphism

Class-based polymorphism allows objects to respond differently to the same member function call. With Class-based polymorphism, and its associated *virtual functions,* it is easy to design *extensible* programs. Class-based polymorphism gives programs a simplified structure.

NOTE: Polymorphism promotes easy debugging, testing and sustainable program maintenance.

Class-based polymorphism is achieved through a simple mechanism – the ‘virtual’ keyword. A member function is marked as polymorphic by the ‘virtual’ keyword preceding its declaration. Such a member function is known as a virtual function, it can now be redefined in all derived classes.

When a virtual function is called, the call is based on the owner object type rather than the type of the invoking pointer or reference.

Consider an example where you have a base class ‘Car’ and a couple of derived classes such as ‘Beetle’, ‘Golf’, ‘Jetta’, etc. Say each of these classes has an ‘accelerate’ function, a different function for each of them. When calling each ‘accelerate’ function, it is clearly beneficial to treat these similar functions generically as objects of the base class ‘Car’. To achieve this functionality, you declare accelerate in the base class as ‘`virtual`’, followed by overriding of the ‘`accelerate`’ function in each of the derived classes. This ‘`accelerate`’ function can look something like this:

virtual void accelerate();

> **NOTE:** Virtual functions allow a program to dynamically (at run time) determine which derived class to use.

5.3.1 Ad-Hoc Polymorphism

Ad-hoc polymorphism is a form of class-based polymorphism where a finite range of types are dealt with. It is nothing more than the overloading of member functions to take different types with the same name. Ad-hoc polymorphism gives us an easy way to, say, overload the binary subtraction operator (‘`-`’) to compare two strings.

5.3.2 Parametric Polymorphism (Generics)

Parametric Polymorphism, also known as *generic programming,* allows your member functions to work with any type of parameters.

In C++, this generic mechanism is exercised through the use of *templates*. Templates allow us to specify overloaded functions (called *template functions*). Templates further facilitate the specification of a related range of classes (called *template classes*).

A template’s definition always begins with the template keyword. The function template’s *formal type parameters* are enclosed within angular brackets (‘< >’) with the ‘`template`’ keyword always followed by these brackets.

Formal type parameters are built-in types used for the specification of function argument types, the return type of a function or for the declaration of variables. Formal type parameters can be preceded by either the ‘`typename`’ or ‘`class`’ keyword.

A function template can be defined as follows:

```
template <class T>
T difference(T val1, T val2)
{
     T total;

     if (val1 > val2)
```

```
          total = val1 – val2;
     else
          total = val2 – val1;

     return total;
}
```

The code given here is incorporated into Program 5.2.

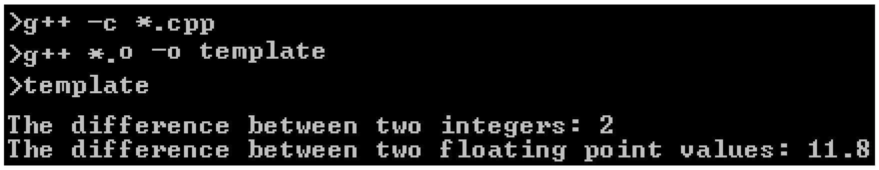
The ‘`difference`’ function declares a single type parameter ‘T’ as the data type to work with. The true type of the data sent to ‘`difference`’ is substituted for ‘T’ throughout the function – the compiler is responsible for this process (it creates a function for the specific data type the function was called for).

Program 5.3 – Function Templates

```
/*
=================
Templ.cpp
- A program making use of function templates
=================
*/

#include<iostream>

using namespace std;

/*
=================
difference template function
- main program
=================
*/
template <class T>
T difference(T val1, T val2)
{
	T total;

	if (val1 > val2)
		total = val1 - val2;
	else
		total = val2 - val1;

	return total;
}

/*
=================
main function
- main program
=================
*/
int main()
```

```
{
	int value1i = 1;
	int value2i = 3;

	float value1f = 7.5;
	float value2f = 19.3;

	cout << "The difference between two integers: " << difference(value1i, value2i)
<< endl;
	cout << "The difference between two floating point values: " <<
difference(value1f, value2f) << endl;

	return 0;
}
//EOF
```

Compilation and Output



```
>g++ -c *.cpp
>g++ *.o -o template
>template
The difference between two integers: 2
The difference between two floating point values: 11.8
```

Class templates, on the other hand, are referred to as *parameterised types* due to their dependency on one or more type parameters for the specification of a precise template class. The class ‘`Draw`’, can, for example, become the basis for many ‘`Draw`’ classes through the use of templates (such as a ‘`Draw`’ class defined for circles, one for arches, etc). The compiler, as with template functions, generates the source code for the template class as required by the programmer.

A template class definition is preceded by the header:

```
template <class T>
```

This indicates the type of the ‘`Draw`’ class to be created (need not be identifier ‘`T`’).

Our ‘`Draw`’ class template might look something like this:

```
template <class T>
class Draw
{
public:
     Draw(); //default constructor
     ~Draw(); //destructor
     create(T&); //construct object based on an element
private:
     int size;
     bool transparent;
};
```

The implementation of this class's member functions will also be preceded by the

```
template <class T>
```

header as shown here:

```
template <class T>
Draw<T>::Draw()
{
     size = 0;
     transparent = false;
}

template <class T>
Draw<T>::create(T &geometric_object)
{
     ...
}
```

When instantiating the object in the main section of the program, you'll specify the type in angular brackets ('< >') followed by the object name, for example:

```
Draw<int> geometricObject();
```

You can now access all the member functions like you'd normally do, by, for instance, writing:

```
geometricObject.Draw();
```

Chapter 6

DATA STRUCTURES: ARRAYS

6.1 Preview

Data structures are specific building blocks used to store data for efficient retrieval and use. There are numerous types of data structures, from *linear* data structures such as lists, arrays and vectors to *graph*-based data structures (data structures consisting of vertices or nodes connected by edges or lines) such as trees and adjacency lists. This chapter mainly focuses on arrays.

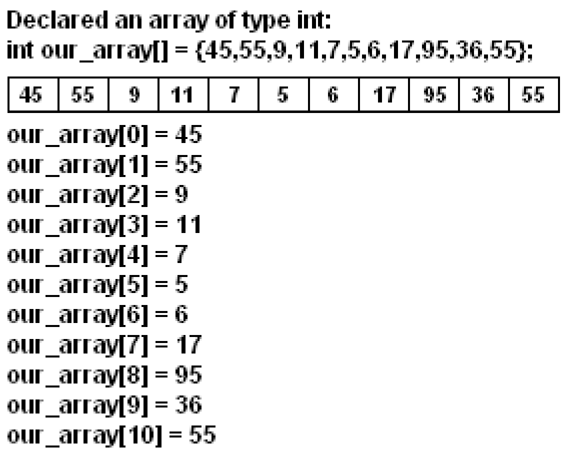
6.2 Arrays

An array is one of the simplest and most frequently used data structures. They are constructed using a sequence of memory positions. These memory positions are related and the array elements are stored adjacent to each other.

An array has a name and type with the different array elements indicated by position numbers contained within square brackets (‘`[]`’). The position number of the first element in an array is always zero (‘`0`’). These positions are called *array subscripts*.

NOTE: Array subscripts are always integer.

Figure 6.1 depicts the creation of an array called ‘`our_array`’. Shown is its memory representation and logical contents as accessed via subscripts.


Declared an array of type int:
int our_array[] = {45,55,9,11,7,5,6,17,95,36,55};

45	55	9	11	7	5	6	17	95	36	55

```
our_array[0] = 45
our_array[1] = 55
our_array[2] = 9
our_array[3] = 11
our_array[4] = 7
our_array[5] = 5
our_array[6] = 6
our_array[7] = 17
our_array[8] = 95
our_array[9] = 36
our_array[10] = 55
```

Fig 6.1 A graphical representation of an array

Subscripts are used for more than just the accessing of array elements, specifically; they can be used in so-called subscript operations, for example:

```
int a, b, c;

a = 4; b = 1;

/* the value 4 contained by our_array[5] now becomes 7 */
our_array[a+b] += 2;

/* array values are also easily assignable to variables */
c = our_array[3]; //c will now equal 11

/* numerous variants of these operations are possible */
c = our_array[6]/2; //c will now equal 3
```

The type of element and the number of elements within each array are specified during declaration of the array.

More examples of array declarations:

```
int numbers[50]; //reserves memory for 50 numbers
char names[30]; //reserves memory for 30 characters

int set[5] = {1, 7, 15, 8, 13}; //declare and initialise
int set2[] = {1, 5, 7, 9}; //set2[3] will be 9
int set3[10] = {0}; //initialise all elements to 0
```

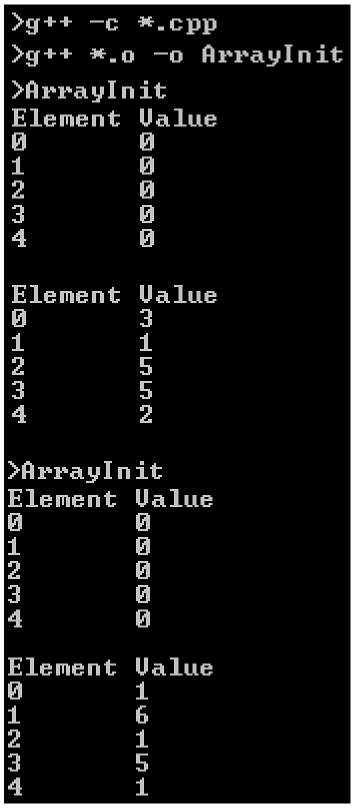
NOTE: When you initialise an array during its declaration, it is not required to indicate the number of elements.

Program 6.1 – Basic Array Initialisations

```
/*
=================
ArrayInit.cpp
- A program used for the initialisation of an array via a for loop
=================
*/

#include<iostream>
/* for our random function */
#include<time.h>
#include<stdlib.h>

using namespace std;

/*
=================
main function
- main program
=================
*/
int main()
{
        int our_array[5];  //declare our array to contain 5 integers

        /* a for loop used to initialise our array to 0 */
        for(int i = 0; i <= 4; i++)
        {
                our_array[i] = 0;
        }

        cout << "Element" << " Value" << endl;

        /* use another for loop to print the array elements */
        for(int j = 0; j <= 4; j++)
        {
                cout << j << "\t" << our_array[j] << endl;
        }

        /* makes sure the random values are different each time we run the program */
        srand(time(NULL)); //built in C/C++ seeding function

        /* a for loop used to randomly assign values to the array elements */
        for(int k = 0; k <= 4; k++)
        {
                our_array[k] = rand()%6 +1;  //random values between 1 and 6
        }

        cout << endl << "Element" << " Value" << endl;

        /* loop to print the array elements */
        for(int I = 0; I <= 4; I++)
        {
                cout << I << "\t" << our_array[I] << endl;
        }

        return 0;
}
//EOF
```

Compilation and Output(running the program twice)

Due to the random function your output may differ…



```
>g++ -c *.cpp
>g++ *.o -o ArrayInit
>ArrayInit
Element Value
0       0
1       0
2       0
3       0
4       0

Element Value
0       3
1       1
2       5
3       5
4       2

>ArrayInit
Element Value
0       0
1       0
2       0
3       0
4       0

Element Value
0       1
1       6
2       1
3       5
4       1
```

Arrays can contain any type – even characters. For instance, you can initialise a character array by use of a string:

`char our_string[] = “Hello World!”;`

The string “`Hello World!`” contains 12 characters plus the string termination character (‘`\0`’). All strings terminate with this character. Figure 6.2 illustrates the alternate creation of such a character array:

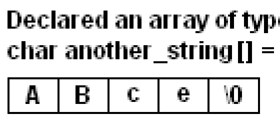

Declared an array of type char:
char another_string [] = {'A', 'B', 'c', 'e'};

Fig 6.2 A graphical representation of a character array.

Character arrays can also be entered from the keyboard:

```
char our_string[10];

cin >> our_string;
```

6.2.1 Sorting of Arrays

Sorting, together with searching, is one of the most integral operations regularly performed on data structures. Many sorting algorithms exist; I will look at bubble sorting, also known as exchange sorting.

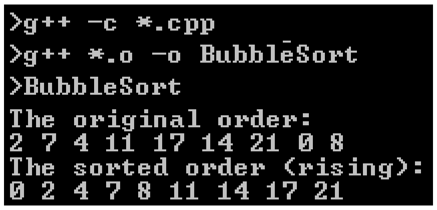
Bubble sorting works by continually stepping through the elements of an array. Two elements are compared during each step with the algorithm swapping these elements whenever they are found to be in the incorrect order. This process continues until all elements are sorted.

Program 6.2 sorts an array in ascending order.

Program 6.2 – Bubble Sort

```
/*
=================
Bubblesort.cpp
- A program implementing bubble sort
=================
*/

#include<iostream>

using namespace std;

/*
=================
main function
- main program
=================
*/
int main()
{
        int our_array[9] = {2, 7, 4 ,11, 17, 14, 21, 0, 8};  //declare and initialise our array
        int temp_store = 0;
19              cout << "The original order:" << endl;

20              /* print the contents of the original array */
21              for(int i = 0; i <= 8; i++)
22              {
23                      cout << our_array[i] << " ";
24              }

25              /* sort the array */
26              for(int step = 1; step <= 8; step++)
27              {
28                      for(int k = 0; k <= 7; k++)
29                      {
30                              if(our_array[k] > our_array[k+1])  //compare
31                              {
32                                      temp_store = our_array[k];  //exchange
33                                      our_array[k] = our_array[k+1];
34                                      our_array[k+1] = temp_store;
35                              }
36                      }
37              }

38              cout << endl << "The sorted order (rising):" << endl;

39              /* print the contents of the sorted array */
40              for(int j = 0; j <= 8; j++)
41              {
42                      cout << our_array[j] << " ";
43              }

44              return 0;
45      }
46      //EOF
```

Compilation and Output



```
>g++ -c *.cpp
>g++ *.o -o BubbleSort
>BubbleSort
The original order:
2 7 4 11 17 14 21 0 8
The sorted order (rising):
0 2 4 7 8 11 14 17 21
```

Bubble Sorting

The bubble sort algorithm is implemented from lines 26 to 37. The given algorithm's steps can be summarised as the comparison between adjacent elements. It swaps the adjacent elements whenever the first element is greater than the second. This is done for each pair of adjacent elements and the process is repeated until the last element is reached. The process is thus repeated for one element less during each step.

The ‘`temp_store`’ variable is used for the temporary storage of array elements.

Bubble sorting is highly inefficient when it has to deal with a large collection of elements. It must not be considered anything more than a simple algorithm, one that’s easy to implement and understand.

To improve performance, you can reverse the array traversal order. This can be done during each step by passing from top to bottom and then from bottom to top, alternately. This bubble sort variant is known as *bidirectional bubble sort* or *shuttle sort*.

6.2.2 Array Searching

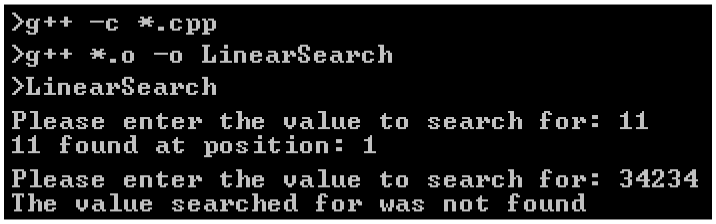
Searching is an essential operation for effective array use. Finding some key value in an array is often critical for many text parsing applications. This section deals with both linear and binary searching techniques. *Linear search* is a simple technique often used for the searching of small or unsorted arrays. *Binary search* is a high-speed technique often used for big, sorted arrays.

Linear search works by comparing each element of the array with the value being searched for. Program 6.3 implements linear searching.

Program 6.3 – Linear Search

```
/*
=================
Linearsearch.cpp
- A program implementing the linear search technique
=================
*/

#include<iostream>

using namespace std;

#define SIZE 10  //define the constant SIZE

/*
=================
main function
- main program
=================
*/
int main()
{
        int our_array[SIZE] = {55, 11, 22, 19, 71, 48, 37, 90, 211, 5};
        int position, search_value;

        cout << endl << "Please enter the value to search for: ";
21          cin >> search_value;

22          /* linear search algorithm */
23          for(int i = 0; i <= SIZE - 1; i++)
24          {
25                  if(our_array[i] == search_value)
26                  {
27                          position = i;
28                          break;
29                  }
30                  else
31                          position = -1;
32          }

33          if(position == -1)
34                  cout << "The value searched for was not found" << endl;
35          else
36                  cout << search_value << " found at position: " << position << endl;

37          return 0;
38  }
39  //EOF
```

Compilation and Output



```
>g++ -c *.cpp
>g++ *.o -o LinearSearch
>LinearSearch
Please enter the value to search for: 11
11 found at position: 1
Please enter the value to search for: 34234
The value searched for was not found
```

Linear Searching

The program does a sequential search through the array, in the process checking every element until a match is found (lines 23 to 31). The position of the matching element is returned, otherwise ‘`-1`’.

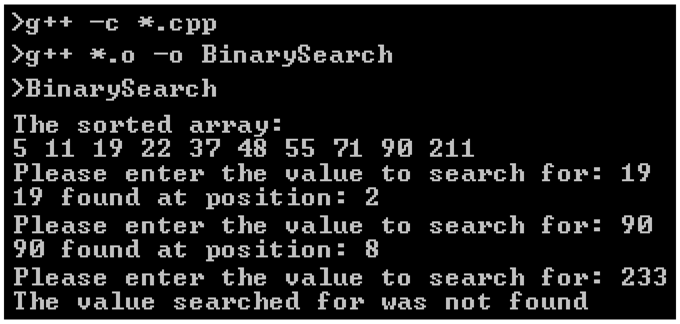
The next program deals with binary search. *Binary search* works by finding the median in a set of values (the middle value in the sorted array); this median is used as a kind of divider to determine whether a value comes before or after it. Say the desired value falls in the lower half of the median; then this half is divided by a new median. The search repeatedly checks to see whether a searched for value lies in the lower or upper half of a median, continuously narrowing the search area until the value is found or the interval is empty.

NOTE: After each comparison, binary search eliminates one half of the elements in an array.

Program 6.4 – Binary Search

```
/*
=================
Binarysearch.cpp
- A program implementing the binary search technique
=================
*/

#include<iostream>

using namespace std;

#define SIZE 10//define the constant SIZE

/*
=================
main function
- main program
=================
*/
int main()
{
	int our_array[SIZE] = {55, 11, 22, 19, 71, 48, 37, 90, 211, 5};
	int position, search_value, median, low, high;
	int temp_store; //for our sorting algorithm

	low = 0; //the position in the array from where we start our search
	high = SIZE; //the final position in our array

	/* first sort the array for binary search to work */
	for(int step = 1; step <= SIZE - 1; step++)
	{
		for(int k = 0; k <= SIZE - 2; k++)
		{
			if(our_array[k] > our_array[k+1])  //compare
			{
				temp_store = our_array[k];  //exchange
				our_array[k] = our_array[k+1];
				our_array[k+1] = temp_store;
			}
		}
	}

	/* print the contents of the sorted array */
	cout << endl << "The sorted array:" << endl;

	for(int j = 0; j <= SIZE - 1; j++)
	{
		cout << our_array[j] << " ";
	}
42          cout << endl << "Please enter the value to search for: ";
43          cin >> search_value;

44          /* binary search algorithm */
45          while(low <= high)
46          {
47                  median = (low + high)/2; //start exactly in the middle

48                  if(search_value == our_array[median])
49                  {
50                          position = median; //position found!
51                          break;
52                  }
53                  else
54                          if(search_value < our_array[median])
55                          {
56                                  high = median - 1; //search lower half
57                          }
58                          else
59                                  low = median + 1; //search upper half

60                          position = -1; //didn't find it
61          }

62          /* print the results */
63          if(position == -1)
64                  cout << "The value searched for was not found" << endl;
65          else
66                  cout << search_value << " found at position: " << position << endl;

67          return 0;
68  }
69  //EOF
```

Compilation and Output



```
>g++ -c *.cpp
>g++ *.o -o BinarySearch
>BinarySearch
The sorted array:
5 11 19 22 37 48 55 71 90 211
Please enter the value to search for: 19
19 found at position: 2
Please enter the value to search for: 90
90 found at position: 8
Please enter the value to search for: 233
The value searched for was not found
```

Binary Searching

The binary search algorithm is defined from lines 45 to 61. This section of code compares the middle element of the array with the value being searched for; if they are equal, the value is found (line 50), if the element being searched for is less than this middle element, the first half of the array is searched (line 56) else the second half is searched (line 59).

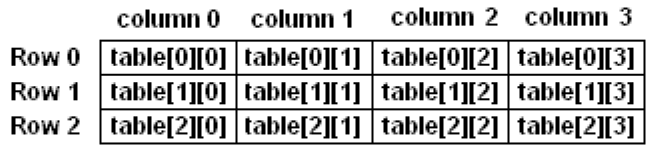
6.2.3 Multi-dimensional Arrays

The arrays illustrated thus far have all been indexed via single integers (‘`our_array[3]`’). *Multi-dimensional arrays* are indexed using multiple integers (‘`our_array[1][4]`’). These arrays, indexed in rows and columns and thus constituting a table, are commonly referred to as *two-dimensional arrays*.

Declaration of a two-dimensional array:

```
int table[3][4];
```

This array, ‘`table`’, consists of three rows and four columns. Figure 6.3 shows the logical table-like structure of such a 2D array.



	column 0	column 1	column 2	column 3
Row 0	table[0][0]	table[0][1]	table[0][2]	table[0][3]
Row 1	table[1][0]	table[1][1]	table[1][2]	table[1][3]
Row 2	table[2][0]	table[2][1]	table[2][2]	table[2][3]

Fig 6.3 Graphical representation of a table constructed via a two-dimensional array

You now finally have enough knowledge to write your first game (what it’s really all about!). This game will be a simple maze traversal, but expanding it to a full sized text based adventure won’t take too much effort.

Perhaps after mastering basic OpenGL or Direct3D topics, you can come back to this program and make something more graphical. I, for example, used Program 6.5 as the basis for a Pocket PC game, the image below shows this game, *Catacomb Commander*, in action:

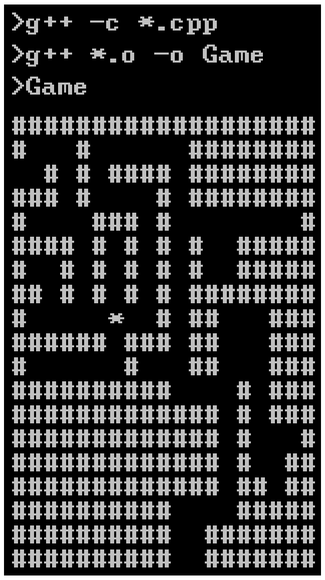
Program 6.5 – The First Game (making use of a two-dimensional array)

```
/*
===============
Catacomb Commander
CataEngine.h
===============
*/

#include<iostream>
#include<conio.h> //for getch
#include<dos.h> //for textcolor
#include<stdio.h> //for fileinput

using namespace std;

#define COLUMNS 19 //columns
#define ROWS 19 //rows

/*
===============
CLASS DECLARATIONS
===============
*/

class CEngine
{
public:
        CEngine();
        ~CEngine();
        void PrintMaze(int [][COLUMNS], int);
        void Control(int map[][COLUMNS], int, int);
};
//EOF
```

```
/*
===============
Catacomb Commander
CataEngine.cpp
===============
*/

#include"CataEngine.h"

/*
======================
CEngine
-Empty Default Constructor
======================
*/
CEngine::CEngine()
{
}

/*
======================
~CEngine
-Empty Destructor
======================
*/
CEngine::~CEngine()
{
}

/*
======================
Control
-Game Movement
======================
*/
void CEngine::Control(int map[][COLUMNS], int x, int y) //via number keypad
{
        int user_input;
        map[x][y] = 3;
        PrintMaze(map,COLUMNS); //prints map on each cycle
        map[x][y] = 0;
        /* use getch built in function to grap input from command line */
        user_input = getch();

        /* recursively analyse user input */

        switch(user_input)
        {
          case '6':
                        if(map[x][y+1] == 0)
                                Control(map,x,y+1);     //move right
                                else
                                        if(map[x][y+1] == 1)
                                                Control(map,x,y); //don't move
                break;

                case '4':  //move left
                if(map[x][y-1] == 0)
                    Control(map,x,y-1); //move left
                else
                    if(map[x][y-1] == 1)
                        Control(map,x,y); //don't move
        break;

        case '2':
                if(map[x+1][y] == 0)
                    Control(map,x+1,y); //move down
                else
                    if(map[x+1][y] == 1)
                        Control(map,x,y); //don't move
        break;

        case '8':
                if(map[x-1][y] == 0)
                    Control(map,x-1,y); //move up
                else
                    if(map[x-1][y] == 1)
                        Control(map,x,y); //don't move
        break;

        case 'x': case 'X': break;
        default :
                Control(map,x,y); //don't move
    }
}

/*
=====================
PrintMaze
-Draws the Level Map
=====================
*/
void CEngine::PrintMaze(int maze[][COLUMNS], int size)          //map creator
{
    /* clear the screen by printing 25 newlines */
    for(int k = 0; k <= 25; k++)
        cout << endl;

    /* draw the map and player */
    for(int i = 0; i <= size - 1; i++)
    {
        for(int j = 0; j <= size - 1; j++)
        {
            if (maze[i][j] == 3)
                cout << "*";  //draw player
            else
            if(maze[i][j] == 1)
                cout <<"#";       //draw walls
            else
                cout << " ";  //draw halls
        }
        cout << endl;
    }
}
```

```
//EOF

/*
===============
Catacomb Commander
To play: 2 = down, 6 = right ,8 = up, 4 = left
===============
*/

#include"CataEngine.h"

int main()
{
        /* init the map - a 2D array */
        int map[ROWS][COLUMNS] =  {{1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1},
                                   {1,0,0,0,1,0,0,0,0,0,0,1,1,1,1,1,1,1,1,1},
                                   {0,0,1,0,1,0,1,1,1,1,0,1,1,1,1,1,1,1,1,1},
                                   {1,1,1,0,1,0,0,0,0,1,0,1,1,1,1,1,1,1,1,1},
                                   {1,0,0,0,0,1,1,1,0,1,0,0,0,0,0,0,0,0,0,1},
                                   {1,1,1,1,0,1,0,1,0,1,0,1,0,0,1,1,1,1,1,1},
                                   {1,0,0,1,0,1,0,1,0,1,0,1,0,0,1,1,1,1,1,1},
                                   {1,1,0,1,0,1,0,1,0,1,0,1,1,1,1,1,1,1,1,1},
                                   {1,0,0,0,0,0,0,0,0,1,0,1,1,0,0,0,1,1,1,1},
                                   {1,1,1,1,1,1,0,1,1,1,0,1,1,0,0,0,1,1,1,1},
                                   {1,0,0,0,0,0,0,1,0,0,0,1,1,0,0,0,1,1,1,1},
                                   {1,1,1,1,1,1,1,1,1,1,1,0,0,0,0,1,0,1,1,1},
                                   {1,1,1,1,1,1,1,1,1,1,1,1,1,1,0,1,0,1,1,1},
                                   {1,1,1,1,1,1,1,1,1,1,1,1,1,1,0,1,0,0,0,1},
                                   {1,1,1,1,1,1,1,1,1,1,1,1,1,1,0,1,0,0,1,1},
                                   {1,1,1,1,1,1,1,1,1,1,1,1,1,1,0,1,1,0,1,1},
                                   {1,1,1,1,1,1,1,1,1,1,1,0,0,0,0,1,1,1,1,1},
                                   {1,1,1,1,1,1,1,1,1,1,1,0,0,1,1,1,1,1,1,1},
                                   {1,1,1,1,1,1,1,1,1,1,1,0,0,1,1,1,1,1,1,1}};

        int x = 2, y = 0; //starting position on the map ROW 2, COLUMN 0

        CEngine instance;
        instance.Control(map,x,y);
}
//EOF
```

Compilation and Output



```
>g++ -c *.cpp
>g++ *.o -o Game
>Game
###################
#   #      ########
  # # #### ########
### #    # ########
#    ### #        #
#### # # # #  #####
#  # # # # #  #####
## # # # # ########
#     *  # ##   ###
###### ### ##   ###
#      #   ##   ###
##########    # ###
############# # ###
############# #   #
############# #  ##
############# ## ##
##########    #####
##########  #######
##########  #######
```

<u>2D-Arrays</u>

The multiple-dimensional array is initialised on line 140 to line 158. This array, ‘`map`’, is the “level” the player gets to walk around in. After setting up the starting position (line 159), I call the member function, ‘`Control(map,x,y)`, with this position and the array as parameters.

The ‘`Control`’ member function starts off by calling the ‘`PrintMaze`’ member function. The ‘`Control`’ member function also analyses the input via the numerous case statements, recursively progressing until the player exits the maze or the ‘`x`’ key is pressed.

…Nihil Sine Labore…