1 Table of Contents

2 C++ Basics

2.1 Overview

C++ is a **statically typed** (type checking is performed at compile time as opposed to run-time; the program will be checked for type errors during compilation), **compiled**, **general-purpose**, **case-sensitive**, **free-form** programming language that supports **procedural**, **object-oriented**, and **generic programming**.

C++ is regarded as a **middle-level** language, as it comprises a combination of both high-level and low-level language features.

C++ is a **superset** of C, and therefore virtually any legal C program is a legal C++ program (with a few notable exceptions, such as the use of the 'new' or 'class' keywords). Whereas in C#, everything is inside classes (including the Main() method) this is not the case in C++. However, C++ fully supports object-oriented programming as well.

C++ consists of:

- The core language and building blocks.
- In-built **C++ Standard Library**, with a rich set of functions manipulating files, strings etc.
- In-built **Standard Template Library (STL)**, with rich set of methods manipulating data structures etc.

The **ANSI standard** ensures that C++ is portable – it can be compiled on any platform.

The primary interfaces of PC Windows and Apple OSX are written in C++.

2.2 Basic Syntax

A C++ program is defined as a collection of **objects** that communicate with each other by invoking each other's **methods**. The program consist of **Classes**, **Objects** (instances of classes), **Methods** and **Instant Variables** (internal class variables that define the State of a class instance).

```
#include <iostream>  // iostream header is used (input output operations)
using namespace std; // using collection of classes std
// main() is where program execution begins.
int main()
{
   cout << "Hello World" << endl; // prints Hello World
                                  // endl better than "Hello World\n", as \n is counted as
                                  // an extra character, and endl will flush
                                  // the stream (the i/o stream to screen, file etc.)
   return 0;
}
```

A C++ **identifier** is a name used to identify a variable, function, class, module, or any other user-defined item. An identifier starts with a letter A to Z or a to z or an underscore (_) followed by zero or more letters, underscores, and digits (0 to 9). Identifiers are case-sensitive.

2.3 Comments

Program comments are explanatory statements that you can include in the C++ code that you write and helps anyone reading it's source code.

```
/* Comment out printing of Hello World:
cout << "Hello World"; // prints Hello World
*/
```

2.4 Data Types

Type	Keyword
Boolean	bool
Character	char
Integer	int
Floating point	float
Double floating point	double
Valueless	void
Wide character	wchar_t

Several of the basic types can be modified using one or more of these type modifiers:

- signed
- unsigned
- short
- long

Type	Typical Bit Width	Typical Range
char	1byte	-127 to 127 or 0 to 255
unsigned char	1byte	0 to 255
signed char	1byte	-127 to 127
int	4bytes	-2147483648 to 2147483647
unsigned int	4bytes	0 to 4294967295
signed int	4bytes	-2147483648 to 2147483647
short int	2bytes	-32768 to 32767
unsigned short int	Range	0 to 65,535

signed short int	Range	-32768 to 32767
long int	4bytes	-2,147,483,648 to 2,147,483,647
signed long int	4bytes	same as long int
unsigned long int	4bytes	0 to 4,294,967,295
float	4bytes	+/- 3.4e +/- 38 (~7 digits)
double	8bytes	+/- 1.7e +/- 308 (~15 digits)
long double	8bytes	+/- 1.7e +/- 308 (~15 digits)
wchar_t	2 or 4 bytes	1 wide character. Long version of standard ASCII characters (8-bit)

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

int main()
{
   cout << "Size of char : " << sizeof(char) << endl;
   cout << "Size of int : " << sizeof(int) << endl;
   return 0;
}
```

2.5 Typedef

You can create a new name for an existing type using **typedef**:

```
typedef int feet;
feet distance;
```

2.6 Enum Data Type

An **enumerated** (enum) type declares an optional type name and a set of zero or more identifiers that can be used as values of the type. Each enumerator is a constant whose type is the enumeration.

```
enum color { red, green, blue } c;
c = blue;
```

By default, the value of the first name is 0, the second name has the value 1, the third has the value 2, and so on. But you can give a name a specific value by adding an initializer:

```
enum color { red, green=5, blue };
```

C++ also allows to define various other types of variables like **Pointer, Array, Reference, Data structures,** and **Classes**.

2.7 Lvalues and Rvalues

There are two kinds of expressions in C++:

lvalue : Expressions that refer to a memory location is called "lvalue" expression. An lvalue may appear as either the left-hand or right-hand side of an assignment.

rvalue : The term rvalue refers to a data value that is stored at some address in memory. An rvalue is an expression that cannot have a value assigned to it which means an rvalue may appear on the right- but not left-hand side of an assignment.

Variables are lvalues and so may appear on the left-hand side of an assignment. Numeric literals are rvalues and so may not be assigned and can not appear on the left-hand side.

2.8 Constants/Literals

Literals are numbers or text assigned directly by the user – in x = 10; 10 is the literal. Literals can be divided into Integer Numerals, Floating-Point Numerals, Characters, Strings and Boolean Values.

An **integer literal** can be a decimal, octal, or hexadecimal constant. A prefix specifies the base or radix: 0x or 0X for hexadecimal, 0 for octal, and nothing for decimal.

A **floating-point** literal has an integer part, a decimal point, a fractional part, and an exponent part. You can represent floating point literals either in decimal form or exponential form.

Boolean literals have two values: true or false.

Character literals are enclosed in single quotes.

String literals are enclosed in double quotes.

Constants can be defined using the `const` keyword and any of the literals, and cannot be altered during the execution of the program. Note that variables declared using the `const` keyword have internal linkage by default (can only be accessed within the same file); unless the `extern` keyword is used (see below for definition of `extern` and global variables).

The **#define** preprocessor directive can be used to define a constant value. In this case, the constant is defined before compile-time.

```
#define LENGTH 10
#define WIDTH  5
#define NEWLINE '\n'
```

It is good programming practice to define constants in capital letters.

2.9 Access Modifiers

2.9.1 Extern Keyword

The `extern` keyword is used to implement 'global' variables in C – i.e. variables that can be accessed and changed from any module (file), as long as they are defined in one of the files, and a header file with the extern declaration is included in all files (in the definition file as well, to ensure that the definition and the declaration are consistent).

Using `extern int x;` tells the compiler that an object of type `int` called `x` exists *somewhere*. It's not the compiler's job to know where it exists, it just needs to know the type and name so it knows how to use it. Once all of the source files have been compiled, the **linker** (program that links variables to their respective references after compilation) will resolve all of the references of `x` to the one definition that it finds in one of the compiled source files. For it to work, the definition of the `x` variable needs to have what's called "external linkage", which basically means that it needs to be declared outside of a function (at what's usually called "the file scope") or in a header file, and without the `static` keyword (global variables are extern by default).

Example Program using External keyword:

Header File (header.h):

```
#ifndef HEADER_H
#define HEADER_H

// any source file that includes this will be able to use "global_x"
extern int global_x;

void print_global_x();

#endif
```

Source File 1:

```
#include "header.h" // include header file with extern declaration and print_global_x()
                    // function declaration

int main()
{
    //set global_x here:
    global_x = 5;

    print_global_x();
}
```

Source File 2:

```
#include <iostream>
#include "header.h" // include header file with extern declaration

void print_global_x()
{
    //print global_x here:
    std::cout << global_x << std::endl;
}
```

Extern Definition from Wikipedia:

The C language (C, C++, C#) does not have a `global` keyword. However, variables declared outside a function have "file scope," meaning they are visible within the file. Variables declared with file scope are visible between their declaration and the end of the compilation unit (`.c` file) (unless shadowed by a like-named variable in a nearer scope, such as a local variable); and they implicitly have external linkage and are thus visible to not only the `.c` file or compilation unit containing their declarations but also to every other compilation unit that is linked to form the complete program. Note that *not* specifying `static` is the same as specifying `extern`: the default is external linkage. External linkage, however, is not sufficient for such a variable's use in other files: for a compilation unit to correctly access such a global variable, it will need to know its type. This is accomplished by declaring the variable in each file using the `extern` keyword (it will be *declared* in each file but may be *defined* in only one; its value can only be set in one of the files). Such `extern` declarations are often placed in a shared header file, since it is common practice for all .c files in a project to include at least one `.h` file.

In general, if we want to use a global variable in the current file only, it is good practice to use the static keyword. If we want to use it in different files, create a definition of the variable in one of the files, and include a header file with a corresponding extern declaration in all files.

2.9.2 Global Functions

To access a function in File 2 from a function in File 1, the function prototype must be included in the file scope of File 1; or a header file with the function prototype must be included in File 1 (and File 2 where the function is defined).

2.9.3 Static Keyword

The **static** keyword has several uses in C++:

- Declaring static variables in functions: the static variables will persist for the ‘lifetime’ of the program (as opposed to **local variables**, which only persist for the duration of the function that they are declared in). Whenever the function is called again, the variable will still contain the last value assigned to it.
- Limiting the scope of ‘global’ variables: a variable declared at the top of all functions without the static keyword, is automatically considered ‘global’ or externally linked, and can be accessed from another file, if both files contain the variable declaration with the extern keyword (or include a header file with the declaration). However, if such as variable is declared as static, this automatically negates the default external linkage of the variable (and including the extern declaration would produce an error). The variable then only has a ‘file scope’ (internal linkage), meaning that it can no longer be accessed from other files using the extern keyword. Static variables are initialized automatically to 0, but it is bad practice to leave them unassigned.
- Limiting the scope of ‘global’ functions: if a function is declared as static, it cannot be declared and used in a different project file.
- Declaring static variables inside a class definition: the value of the variables will persist in different instances of the class, and the variable doesn’t even require an instance of the class in order to exist. Importantly, it is good syntax to refer to static member functions through the use of a class name (class_name::x; rather than instance_of_class.x;). To access the static member, you use the **scope operator**, **::**, when you refer to it through the name of the class. Note that static class variables must be initialized (a value needs to be assigned), but cannot be initialized inside the class.

- Static member functions of a class. Static member functions are functions that do not require an instance of the class, and are called the same way you access static member variables: with the class name rather than a variable name (e.g. a_class::static_function(); rather than an_instance.function();). Static member functions can only operate on static members, as they do not belong to specific instances of a class.

```
class user
{
  private:
  int id;
  static int next_id;   //static variable declaration (initialized outside class)

  public:
  static int next_user_id()
  {
    next_id++;          //increment static variable
    return next_id;     //return static variable
  }
  /* More stuff for the class user */
  user()
  {
    id = user::next_id++; // increment static variable on object construction
  }
};

int user::next_id = 0; //must include type of static variable when setting it
```

2.9.4 Local and Global Variable Initialisation

Local variables are declared within function or block of code.

Global variables are declared outside of functions, usually at the top of the program.

When a local variable is defined, it is not initialized by the system, you must initialize it yourself Global variables are initialized automatically by the system when you define them as follows:

Data Type	Initializer
int	0
char	'\0'
float	0
double	0
pointer	NULL

Moreover, file-scope and function-scope static variables are initialized automatically to 0 (or NULL, which is a #define for 0). If the static variable is an array or struct, all members of the variable are also initialised to 0.

Overall, it is good programming practice to initialize variables properly.

2.9.5 Global Access Summary

If you want to use a variable in multiple files, you should put the declaration of the variable using the extern keyword in one header file, and then include that header file in all source files that need that variable. Then you should put the definition of that variable in one source file that is linked with all the files that use that variable.

If you want to use a function across multiple source files, you should declare the function in one header file (.h) and then put the function definition (prototype) in one source file (.c or .cpp). All code that uses the function should include the .h file.

If you want to use a class in multiple files, you should put the class definition in a header file and define the class methods in a corresponding source file (you an also use inline functions for the methods).

2.9.6 Side Note: Global Access in C#

In C#, which is strictly object-oriented, classes are declared as part of namespaces. If no namespace is declared, Visual Studio will put all classes in the same default namespace. If two different files (in the same project) are using the same namespace, they have access to the class declarations in both files. Classes can be declared as public or internal. If a class is declared as internal, it can only be instantiated in the same file. Classes can also be accessed between different projects (this needs extra code).

Different rules apply to nested classes (classes declared within other classes). In this case, the nested classes (or structs declared within the base class) can be declared as public, protected internal, protected, internal, or private.

2.10 Declaration vs Definition

A declaration provides basic attributes of a symbol: its type and its name. A definition provides all of the details of that symbol--if it's a function, what it does; if it's a class, what fields and methods it has; if it's a variable, where that variable is stored. Often, the compiler only needs to have a declaration for something in order to compile a file into an object file, expecting that the linker can find the definition from another file. If no source file ever defines a symbol, but it is declared, you will get errors at link time complaining about undefined symbols.

2.11 Modifier Types

const	Variables of type **const** cannot be changed by your program during execution. They must be initialized at the same time as they are declared.
volatile	The modifier **volatile** tells the compiler that a variable's value may be changed in ways not explicitly specified by the program.
restrict	A pointer qualified by **restrict** is initially the only means by which the object it points to can be accessed. C++ does not have standard support for `restrict`, but many compilers have equivalents that usually work in both C++ and C, such as the GNU Compiler Collection's and Clang's `__restrict__`, and Visual C++'s `__restrict` and `__declspec(restrict)`.

2.12 Storage Classes

A **storage class** defines the scope (visibility) and life-time of variables and/or functions within a C++ Program. These specifiers precede the type that they modify. There are following storage classes, which can be used in a C++ Program

- **auto**: automatically determine variables type, based on the expression (e.g. `auto c = a+b`, where `a` and `b` are of type `int`, would assign type `int` to `c`).
- **register**: used to define local variables that should be stored in a register instead of RAM. This means that the variable has a maximum size equal to the register size (usually one word) and can't have the unary '&' operator applied to it (as it does not have a memory location). The register should only be used for variables that require quick access such as counters. It should also be noted that defining 'register' does not mean that the variable will be stored in a register. It means that it MIGHT be stored in a register depending on hardware and implementation restrictions.
- **static**: see previous sections.
- **extern**: see previous sections.
- **mutable**: applies only to class objects, which are discussed later in this tutorial. It allows a member of an object to override `constness'. That is, a mutable member can be modified by a const member function.

2.13 Operators

An operator is a symbol that tells the compiler to perform specific mathematical or logical manipulations. C++ is rich in built-in operators and provides the following types of operators:

- Arithmetic Operators (+,-,*,/, ++, -- etc.)
- Relational Operators (<,>,== etc.)
- Logical Operators (&&,||,!)
- Bitwise Operators (&,|, <<,>> etc.)
- Assignment Operators (=, +=, *= etc.)

Misc Operators:

Operator	Description
sizeof	sizeof operator returns the size of a variable in bytes. For example, sizeof(a), where a is integer, will return 4.
Condition ? X : Y	Conditional operator. If Condition is true ? then it returns value X : otherwise value Y
,	Comma operator causes a sequence of operations to be performed. The value of the entire comma expression is the value of the last expression of the comma-separated list.
. (dot) and -> (arrow)	Member operators are used to reference individual members of classes, structures, and unions.
Cast	Casting operators convert one data type to another.
&	Reference operator & returns the address of an variable. For example &a; will give actual address of the variable.
*	Pointer operator * is pointer to a variable.

As in C, C++ has **operator precedence rules**.

2.14 Math Operations and Random Numbers

C++ has a rich set of mathematical operations, which can be performed on various numbers. To utilize these functions you need to include the math header file `<cmath>`. Examples include `cos()`, `sin()`, `tan()`, `log()` and `sqrt()`. To utilize random number generators, include the `<cstdlib>` header file. Random functions include `rand()` and `srand()`.

2.15 Function/Class Libraries, Header Files and Namespaces

As C++ is a mid-level programming language, it contains both **standalone function libraries** (function accessed directly using their names) and **class libraries** with special functionality (parameters and methods; accessed via class instances and static methods). Both types of libraries are located in system **header files,** which can be included in any program that requires a given functionality. Header files are included using the #include <> command.

An example of libraries located in headers are the `<cstdlib>` function library, and the <random> class library. Both libraries contain the functionality required to generate random numbers, but each is accessed differently.

Libraries can be further organised using **namespaces**. Namespaces may contain declarations of variables, functions and classes. A namespace definition begins with the keyword namespace followed by the namespace name as follows:

```
namespace namespace_name {
   // code declarations
}
```

A namespace can be defined in several parts and so a namespace is made up of the sum of its separately defined parts. The separate parts of a namespace can be spread over multiple files. If we want to use a specific namespace within a library, we should place '`using namespace_name`' at the top of our code.

Note that the header files contain only the declarations of the variables, functions and classes, and the corresponding definitions are hidden away in the library source files. These definitions are located by the linker after compilation.

Note that similar access modifier rules apply to variables placed inside namespaces, as they do to variables placed in the global namespace (in general code, without specifying the namespace explicitly). If a variable is declared as external (which it also is by default) in a namespace in a header file, then it will remain global **only in that namespace**. It must then be defined in one of the .cpp files, in the same namespace (with the header file included). If on the other hand, you declare a variable as static inside a .h file (within or without namespace; doesn't matter), and include that header file in various .cpp files, the static variable becomes locally scoped to each of the .cpp files. Every .cpp file that includes that header will have its own copy of that variable.

Similar rules apply to function and class declarations placed inside namespaces of header files. These function and class methods must then be defined in the namespace in one of the .cpp files, with the declaration header file included. Whenever these functions or classes are to be used, the header file is included, and the `using namespace_name` command is placed at the top of the code.

2.16 Loops

While, **For**, **Do While**, **Nested Loops**, **Break**, **Continue**, **Infinite Loops**.

2.17 Decision Making

If, **else**, **nested if else**, **switch**, **nested switch**. The **conditional operator ? :** can be used to replace simple **if...else** statements.

2.18 Arrays

An array stores a fixed-size sequential collection of elements of the same type. An array is used to store a collection of variables of the same type. Arrays are declared as follows:

```
datatype arrayName[arraySize];
```

Arrays are initialized as follows:

```
datatype arrayName[] = {value1,value2,value3,...};
```

Arrays are accessed as follows:

```
double salary = balance[9];
```

It is also possible to declare and initialise **multidimensional arrays**, assign **pointers to arrays**, and pass arrays to functions by pointer.

2.19 Strings

A string is a one-dimensional array of characters which is terminated by a **null** character '\0'. Strings can be declared as follows:

```
char greeting[] = "Hello";
```

The C++ compiler automatically places the '\0' at the end of the string when it initializes the array.

C++ supports a wide range of functions that manipulate null-terminated strings (e.g. `strcpy()`, `strcat()` etc.)

The Standard C++ library also includes a 'string' class, that supports the standard string operations, and additionally provides much more functionality:

```
string str1 = "Hello"; //compiler reads this as string str1 = new string("Hello");
string str2 = "World";
string str3;
str3 = str1 + str2; //compiler recognizes overloaded operator, specified in class methods
```

2.20 Structures

Declared, defined and accessed in the same way as in C (using dot . operator). Structures can be passed to functions by value (entire copies), or by pointer to struct (in which case the entries are accessed using the arrow -> operator). Typedef has a special use with structures, in that the type and entire structure declaration can be assigned to a name:

```
typedef struct
{
   char  title[50];
   char  author[50];
   char  subject[100];
   int   book_id;
} Books;
```

2.21 Functions

Various names: **function**, **method**, **sub-routine** or **procedure**.

The **function definition** contains the main body of the code. It can be placed anywhere in the program (top or bottom of file, different file), as long as the **function declaration** is made available at the top of the code (so the linker can link the declaration to the definition after compilation is finished).

```
return_type function_name(param_type1 param1, param_type2 param2, ...)
{
   //body of the function
}
```

A **function declaration** (**prototype**), tells the compiler about the function name, parameter types, and output type. This should usually be included at the top of the code, before the function is used, in order for the compiler to recognise the function name, output and parameter types (the parameter names can but don't need to be included):

```
return_type function_name(param_type1, param_type2)
```

When you define a function, you can specify a default value for each of the last parameters. This value will be used if the corresponding argument is left blank when calling to the function:

```
int sum(int a, int b=20)
{
  // b will be set to 20 if value not specified in function call (e.g. sum(a)).
  int result;
  result = a + b;
  return (result);
}
```

2.22 Main Function

The `main` function must be declared as a non-member function in the global namespace. This means that it cannot be a static or non-static member function of a class, nor can it be placed in a namespace (even the unnamed namespace).

The name `main` is not reserved in C++ except as a function in the global namespace. You are free to declare other entities named `main`, including among other things, classes, variables, enumerations, member functions, and non-member functions not in the global namespace.

You can declare a function named `main` as a member function or in a namespace, but such a function would not be the `main` function that designates where the program starts.

The `main` function cannot be declared as `static` or `inline`. It also cannot be overloaded; there can be only one function named `main` in the global namespace.

The `main` function cannot be used in your program: you are not allowed to call the `main` function from anywhere in your code, nor are you allowed to take its address.

The return type of `main` must be `int`. No other return type is allowed (this rule is in bold because it is very common to see incorrect programs that declare `main` with a return type of `void`; this is probably the most frequently violated rule concerning the `main` function).

There are two declarations of `main` that must be allowed:

```
int main()               // (1)
int main(int, char*[])   // (2)
```

In **(1)**, there are no parameters.

In **(2)**, there are two parameters and they are conventionally named `argc` and `argv`, respectively. `argv` is a pointer to an array of C strings (which is actually an array of pointers to strings) representing the arguments to the program. `argc` is the number of arguments in the `argv` array.

Usually, `argv[0]` contains the name of the program, but this is not always the case. `argv[argc]` is guaranteed to be a null pointer.

Note that since an array type argument (like `char*[]`) is really just a pointer type argument in disguise, the following two are both valid ways to write **(2)** and they both mean exactly the same thing:

```
int main(int argc, char* argv[]) //second argument is pointer to array of pointers
int main(int argc, char** argv) //second argument is pointer to a pointer
```

Some implementations may allow other types and numbers of parameters; you'd have to check the documentation of your implementation to see what it supports.

`main()` is expected to return zero to indicate success and non-zero to indicate failure. You are not required to explicitly write a `return` statement in `main()`: if you let `main()` return without an explicit `return` statement, it's the same as if you had written `return 0;`. The following two `main()` functions have the same behaviour:

```
int main() { }
int main() { return 0; }
```

2.23 Pointers and References

In C++ it is important to distinguish between **pointers** and **references**. Whereas in C only pointers are used, both the pointer and reference variables can be used in C++.

A reference, like a pointer, is a variable that you can use to refer indirectly to another variable. A reference declaration has essentially the same syntactic structure as a pointer declaration. The difference is that while a pointer declaration uses the * operator, a reference declaration uses the & operator.

For example:

```
int i = 3;
int *ptr_i = &i;
int &ref_i = i;
```

`ptr_i` is a variable of type 'pointer to int', whose initial value is the address of variable `i`.

`ref_i` is a variable of type 'reference to int', whose initial value is the address of variable `i`.

The big difference between pointers and references is that you must use an explicit * operator to dereference a pointer, but you don't use an operator to dereference a reference. Thus, assignments such as:

```
*ptr_i = 4;
ref_i = 4
```

both change the value of `i` to 4.

This difference in appearance is significant when you're choosing between pointers and references for function parameter types and return types. This is especially true in functions that declare **overloaded operators**.

2.23.1 Pointers and References: Overloaded Operators

We want to overload the ++ operator, to be used with the following enum type:

```
enum day {Sunday, Monday, Tuesday, Wednesday}; day x;
```

In this case, the expression `++x` does not compile. If you want it to, you must define a function named `operator++` which accepts `day` as an argument.

Invoking `++x` should change the value in `x`. Therefore, declaring `operator++` with a parameter of type day, as in:

```
day operator++(day d);
```

won't have the desired effect. This function passes its argument by value, which means the function sees a copy of the argument, not the argument itself. For the function to alter the value of its operand, it must pass that operand either by a pointer or by a reference.

Passing the argument by a pointer, as in:

```
day *operator++(day *d);
```

would let the function alter the value of the day by storing the incremented value into `*d`. However, you would then invoke the operator using an expression such as `++&x`, which doesn't look right.

The proper way to define operator++ is with a reference as the parameter type:

```
day &operator++(day &d)
{
d = (day)(d + 1);
return d;
}
```

Using this function, expressions such as `++x` have the proper appearance as well as the proper behaviour.

Passing by reference is not just the better way to write `operator++`, it's the only way. The following declaration will not compile:

```
day *operator++(day *d);
```

Every overloaded operator function must either be a member of a class, or have a parameter of type T, T &, or T const &, where T is a class or enumeration type.

C++ does not let you overload operators that redefine the meaning of operators for built-in types, including pointer types. Thus, you cannot declare:

```
int operator++(int i); //error
```

which attempts to redefine the meaning of ++ for `int`, nor can you declare:

```
int *operator++(int *i); //error
```

which attempts to redefine ++ for `int *`.

2.23.2 Pointers and References: Const Pointers and References

In C++ (and C), you can add a type modifier `const` to a pointer, indicating that the value of the pointer (i.e. the address of the variable it points to) cannot change over it's lifetime; the pointer can only point to that variable. It is important to distinguish constant pointers from pointers to a const (in which case the address in the pointer can be changed):

```
char * const a;
```

`*a` is writable, but `a` is not; in other words, you can modify the value *pointed to* by `a`, but you cannot modify `a` itself; `a` is a *constant pointer to* `char`. `const_cast` can be used to cast away the constness of the pointer in this case, as the variable it points to is not constant.

```
const char * a;
```

`a` is writable, but `*a` is not; in other words, you can modify `a` (pointing it to a new location), but you cannot modify the value *pointed to* by `a`.

In C++, references are inherently declared with the `const` modifier, meaning that they cannot be re-binded once they are bound to a variable; there is no notation in C++ for re-binding a reference. Since you can't change the reference after you bind it, you must bind the reference at the beginning of its lifetime. Otherwise, the compiler will produce an error.

2.23.3 Pointers and References: Null References

Pointers, even constant pointers, can have a value of NULL. In C and C++, NULL is a #define for the an integer constant 0. Thus setting a value to NULL is equivalent to setting it to 0. A null pointer doesn't point to anything. This difference suggests that if you want your function to avoid receiving a NULL pointer, you should use a reference as a parameter type.

2.23.4 Pointers and References: Readability

Passing by pointer allows you to explicitly see at the call site whether the object is passed by value or by reference:

```
func(ref); // Is ref passed by value or by reference? Need to check func() definition.
func2(&ptr); // func2 passes "by pointer" - no need to look up function definition.
```

2.23.5 Returning Pointers and References

In addition to accepting pointers and references as parameters, functions may also return pointers or references. As returning a pointer or reference to an automatic variable, which disappears after a function is complete, would create an undefined behaviour, the variable must either be static inside the function, or be a global/static variable with a file scope.

If the variable that the pointer points to is in a different file, it must be declared as static in the target function. The static variable will persist in the internal buffer, until the function is called again.

2.23.6 Pointers to Arrays, Structures and Classes

Pointers are closely related to arrays, and are therefore commonly used with for array operations. Once the address of the first element of an array is assigned to a pointer, **pointer arithmetic** (*(ptr++),*(ptr--), *(ptr+1), ptr[] etc.) can be used to access the elements of the array.

Passing an array to a function in C and C++ is equivalent to passing the address of the first element of the array (i.e. passing by pointer). The address must be assigned to a pointer in the function arguments. The array can then be accessed using pointer arithmetic (*(ptr+1), *(ptr++), or ptr[]; which the compiler reads as equivalent). Pointers to two-dimensional arrays are declared as follows: **int** ptr** (pointer to pointer), and the array is accessed through the pointer as e.g. ptr[10][8].

It is also possible to declare an **array of pointers**, and **pointers to other pointers** (e.g. int **ptr).

There is also a useful –> operator, used to access the elements of a structure or class, via a pointer to the structure or class.

2.23.7 Pointers to Functions

A function pointer is a variable that stores the address of a function, and allows for the function to be called via the pointer. The pointer can be passed into another function or can be used to set up Callback functions (functions that are invoked when a particular event happens; the callback function is called, and this notifies your code that something of interest has taken place).

The syntax for declaring a function pointer is as follows:

```
output_type (*function_name)(input_type(s));
```

2.24 Date and Time

The C++ standard library inherits data and time manipulation libraries from C. These are located in the <ctime> header file. There are four time-related types: **clock_t, time_t, size_t**, and **tm**, which are returned by the library functions either by value or in the form of a pointer. The functions can also return a pointer to a date/time string. The types clock_t, size_t and time_t are capable of representing the system time and date as some sort of integer. The structure type tm holds the date and time in the form of a C structure having the following elements

```
struct tm {
  int tm_sec;   // seconds of minutes from 0 to 61
  int tm_min;   // minutes of hour from 0 to 59
  int tm_hour;  // hours of day from 0 to 24
  int tm_mday;  // day of month from 1 to 31
  int tm_mon;   // month of year from 0 to 11
  int tm_year;  // year since 1900
  int tm_wday;  // days since sunday int tm_yday;  // days since January 1st
  int tm_isdst; // hours of daylight savings time
}
```

2.25 Basic Input/Output

C++ I/O occurs in **streams**, which are sequences of bytes. If bytes flow from a file or a device like a keyboard, a disk drive, or a network connection etc. to main memory, this is called **input stream** and if bytes flow from main memory to a file or a device like a display screen, a printer, a disk drive, or a network connection, etc, this is called an **output stream**.

The following three headers contain the important input/output libraries in C++: <iostream>, <iomainp> and <fstream>. The <fstream> header contains the services required for used-controlled file processing (input/output to/from files).

2.25.1 Standard Output Stream (cout)

The predefined object `cout` is an instance of the **ostream** class. The cout object is said to be "connected to" the standard output device, which usually is the display screen; although the output stream can be redirected to a file using services from the <fstream> header.

`cout` is used in conjunction with the stream insertion operator, which is written as <<. Explicit instantiation of the `cout` class is not required in this case – the compiler interprets `cout` as an instantiation.

The C++ compiler also determines the data type of variable to be output and selects the appropriate stream insertion operator (<<) to display the value (using operator overloading methods, which also overload each other depending on input type).

`<< endl` is used after cout, which adds a new line to the output. `endl` is a function in the std namespace. Thus the << operator redirects the output stream of the function to the output.

2.25.2 Standard Input Stream (cin)

The predefined object **cin** is an instance of **istream** class. The cin object is said to be attached to the standard input device, which usually is the keyboard. The **cin** is used in conjunction with the stream extraction operator, which is written as >>.

The >> operator is used to assign the input stream to a variable. The C++ compiler also determines the data type of the entered value and selects the appropriate stream extraction operators (overloaded method) to extract the value and store it in the given variables.

The stream extraction operator >> may be used more than once in a single statement. To request more than one datum you can use the following:

```
cin >> name >> age;
```

2.25.3 Standard Error Stream (cerr)

The predefined object **cerr** is an instance of **ostream** class. The cerr object is said to be attached to the standard error device, which is also a display screen but the object **cerr** is un-buffered and each stream insertion to cerr causes its output to appear immediately. **cerr** is also used in conjunction with the stream insertion operator <<.

2.25.4 Standard Log Stream (clog)

The predefined object **clog** is an instance of **ostream** class. The clog object is said to be attached to the standard error device, which is also a display screen but the object **clog** is buffered. This means that each insertion to clog could cause its output to be held in a buffer until the buffer is filled or until the buffer is flushed (done using additional commands). The **clog** is also used in conjunction with the stream insertion operator <<.

2.25.5 cout vs cerr vs clog

cout, cerr and clog are different output streams. Each of the streams can be redirected independently, using the `rdbuf()` class method. In general cout is used for actual program output, clog is used for logging information, and cerr is used for error information. The difference between clog and cerr, is that clog is buffered (can hold input stream without displaying it – done using additional commands), whilst cerr is not (displays input stream immediately).

3 C++ Object Oriented

The main purpose of C++ programming is to add object orientation to the C programming language and classes are the central feature of C++ that supports object-oriented programming and are often called user-defined types.

3.1 Classes and Objects

Objects are instances of classes.

Classes are blueprints of objects, and define what the object will contain (the variables), what operations can be performed on the object (the functions), and the access to these components (using access modifiers). The variables and methods within a class are referred to as the **members** of the class.

3.1.1 Basic Syntax

```
#include <iostream>

using namespace std;

class Line
{
   //Public variable and function declarations (can be accessed from class instance)
   public:
      void setLength(double len);
      double getLength(void);
      Line(double len);  //Class constructor function

   //Private variable and function declarations (accessed only through member functions)
   private:
      double length;
};

/*Member function definitions, including the constructor function, defined outside of the
class declaration, using the scope resolution operator :: */
Line::Line(double len)
{
   cout << "Object is being created, length = " << len << endl;
   length = len; //set private class member to value passed in constructor
}

void Line::setLength(double len)
{
    length = len;
}

double Line::getLength(void)
{
    return length;
}

// Main function
int main()
{
  Line line(10.0); //create class instance and pass value to class constructor

  //call
  cout << "Length of line : " << line.getLength() <<endl;

  // set line length again
  line.setLength(6.0);

  cout << "Length of line : " << line.getLength() <<endl;

  return 0;
}
```

3.1.2 Class Access Modifiers and Inheritance

The following class access modifiers can be used in C++, in order to achieve the desired level of **encapsulation** (how much is hidden from the outside world) and **abstraction** (i.e. defining what can and cannot be interacted with by the user):

- **public**: public members of a class (variables and functions) can be accessed directly from a class instance (i.e. object), using the **direct member access operator** '**.**'.
- **private**: private members of a class can only be accessed via the class member functions (which need to be public in order to be accessed from an instance). **private** members can also be accessed from the member functions of classes declared as **friends** of the class. By default, all class members are private.
- **protected**: protected members of a class can accessed via the class member functions of the class declaring them, or via the member functions of the class that **derives** (and **inherits**) from it. Protected members can also be accessed from the member functions of classes declared as **friends** of the base class or the derived class.

If any variable or function (regardless of access modifiers) is declared as **static**, it can be accessed directly (without creating a class instance), using the scope resolution operator (::). Static functions are only allowed to modify static variables.

```
class foo
{
    private:
        static int i;
};

int foo::i = 0;
```

In this case, it is important that the declaration is placed in a header file, while the definitions are placed in .cpp files. If multiple class instances are created in the same or in different files, the static variable persists in all these instances. If instead we would initialise it in the header file, and the header would be included in multiple files, the linker would not be able to determine which initialisation should be used; this would result in linker errors.

Moreover, class access modifiers can be used in inheritance, in order to define the access to the variable or functions **from within the derived class**. The following example code summarises the use of class access modifiers in C++:

```
class A
{
public:
    int x;
protected:
    int y;
    int func(int);
private:
    int z;
};

class B : public A
{
    // x is public - x can still be accessed from an instance of B
    // y is protected - classes deriving from B still have access to y
    // z is not accessible from B

    /* func is protected, it can only be accessed via public methods in either B or A. func
       still has access to the private variable z, declared in A. */
```

```
};

class C : protected A
{
    // x is protected – classes deriving from C still have access to x
    // y is protected – classes deriving from C still have access to y
    // z is not accessible from C
};


class D : private A
{
    // x is private – classes deriving from D will not have access to x, but D still does
    // y is private – classes deriving from D will not have access to y, but D still does
    // z is not accessible from D
};
```

Multiple inheritance is also supported in C++. It works in exactly the same way as single inheritance does, but uses the following syntax:

```
class derivedClassName: accessSpecifier baseClassA, accessSpecifier baseClassB, ...
```

3.1.3 Member Functions

A member function of a class is a function that has its definition or its prototype within the class definition like any other variable. It operates on any object of the class of which it is a member, and has access to all the members of a class for that object.

Member functions can be defined within the class definition or separately using **scope resolution operator, ::**. Defining a member function within the class definition declares the function **inline**, even if you do not use the inline specifier. So either you can define a function as below:

```
class Box
{
   public:
      double length;      // Length of a box
      double breadth;     // Breadth of a box
      double height;      // Height of a box

      double getVolume(void)
      {
         return length * breadth * height;
      }
};
```

Or, if you like you can declare the function in the class, and define it outside of the class using the **scope resolution operator**, **::** as follows:

```
double Box::getVolume(void)
{
    return length * breadth * height;
}
```

The second method is the recommended one, as class declarations should be separated from member functions definitions – class declarations should be placed in a header file, which can be reused in different files, while the definitions of the member functions should be placed in one of .cpp files.

A public member function can then be called using the dot operator (**.**) on an object as follows:

```
Box myBox;          // Create an object
myBox.getVolume();  // Call member function for the object
```

3.1.4 Constructor and Destructor

A class **constructor** is a special member function of a class that is executed whenever we create new objects of that class (see Basic Syntax section for example). A constructor has the exact same name as the class and it does not have any return type at all, not even void. Constructors can be very useful for setting initial values for certain member variables.

The default constructor does not have any parameters, but if you need, a constructor can have parameters. This helps you to assign initial values to the class variables at the time of its creation. The variables can be initialised inside the constructor function using an **initialization list**:

```
C::C( double a, double b, double c)
{
x = a;
y = b;
z = c;
}
```

or using the following syntax:

```
C::C( double a, double b, double c): x(a), y(b), z(c)
{
  // no need to initialise x, y and z here
}
```

If one class inherits from another, the constructor takes the following from (as example):

```
ClassB(int a=0, int b=0):ClassA(a, b) //passes inputs to constructor of ClassA
```

A **destructor** is a special member function of a class that is executed whenever an object of it's class goes out of scope (e.g. if instance was created only in function, which has now finished execution) or whenever the delete expression is applied to a pointer to the object of that class. A destructor will have exact same name as the class prefixed with a tilde (**~**) and it can neither return a value nor can it take any parameters. The destructor can be very useful for releasing resources before coming out of the program like closing files, releasing memories etc.

3.1.5 Dynamic Memory Allocation

Memory in a C++ program is divided into two parts:

The stack: At compile times, all variable declarations will take up memory on the stack.

The heap: Memory can also be allocated on the heap. This memory can be allocated and released dynamically, i.e. when the program runs.

In C++, the **new** and **delete** keywords are used to allocate memory on the heap. The following syntax is used to allocate memory dynamically for any data type (in this case a double):

```
double *pvalue  = NULL; // Create pointer to double and initialise to NULL
pvalue  = new double;   // Allocate memory on the heap and store address in pointer
*pvalue = 1.1232135242143; // Dereference memory location and assign value
delete(pvalue); //release heap memory
```

When allocating and releasing heap memory for arrays, the syntax is as follows:

```
int* pvalue  = NULL;   // Pointer initialized with null
pvalue  = new int[20]; // Request memory for the variable
pvalue[3] = 10; // equivalent to *(pvalue + 3) = 10;
delete [] pvalue; // release entire array memory associated with pointer

/* Multidimensional arrays */
int** pvalue  = NULL;   // Pointer initialized with null
pvalue  = new int[20][10]; // Request memory for multidimensional array of ints
delete [] pvalue; // release entire array memory associated with pointer
```

When allocating memory for objects, the syntax is as follows:

```
Box* myBox = new Box;
Box* myBoxArray = new Box[4];
```

3.1.6 Copy Constructor

The **copy constructor** is a constructor which creates an object by initializing it with an object of the same class, which has been created previously. The copy constructor is used to:

- Initialize one object from another of the same type.
- Copy an object to pass it as an argument to a function.
- Copy an object to return it from a function.

When copying an object of the same type, only the copy constructor will be called (not the normal constructor). If a copy constructor is not defined in a class, the compiler itself defines one. If the class has pointer variables and has some dynamic memory allocations, then it is a must to have a copy constructor. The most common form of copy constructor is:

```
classname (const classname &obj) {
  // body of constructor }
```

where `obj` is a reference to an existing object, which is used to construct another object of the same type. The use of the copy constructor and dynamic memory allocation are illustrated in the following example:

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

class Line
{
   public:
```

```
      int getLength(void);          // can specify input as void, but don't have to
      Line(int len);                // simple constructor
      Line(const Line &obj);        // the copy constructor
      ~Line();                      // destructor
   private:
      int *ptr;
};

/* Member function definitions including constructor */
Line::Line(int len)
{
    cout << "Normal constructor allocating ptr" << endl;
    ptr = new int; //returns address of heap memory allocation
    *ptr = len;    //dereferencing heap memory (store len in heap memory)
}

Line::Line(const Line &obj)
{
    cout << "Copy memory allocations made in constructor." << endl;
    ptr = new int;
   *ptr = *obj.ptr; // copy the value of len, stored in heap memory, into new heap memory
}

Line::~Line(void)
{
    cout << "Freeing memory!" << endl;
    delete ptr; //release memory in heap
}

int Line::getLength(void)
{
    return *ptr;
}

void display(Line obj)
{
   cout << "Length of line : " << obj.getLength() <<endl;
}

// Main function for the program
int main()
{
   Line line1(10);
   Line line2 = line1; // Call to copy constructor
   return 0;
}
```

3.1.7 Friend Keyword

The following can be defined using the keyword friend: function, member function, class, or class template (see later notes).

A friend function of a class is defined outside that class' scope but it has the right to access all private and protected members of the class. Even though the prototypes for friend functions appear in the class definition, friends are not member functions. To declare a function as a friend of a class, precede the function prototype in the class definition with keyword **friend** as follows:

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

// forward declaration
class B;
class A {
    private:
      int numA;
    public:
      A(): numA(12) { } //same as A() {numA = 12}
      // friend function declaration
      friend int add(A, B);
};

class B {
    private:
       int numB;
    public:
       B(): numB(1) { }
       // friend function declaration
       friend int add(A , B);
};

// Function add() is the friend function of classes A and B
// that accesses the member variables numA and numB
int add(A objectA, B objectB)
{
   return (objectA.numA + objectB.numB);
}

int main()
{
    A objectA;
    B objectB;
    cout<<"Sum: "<< add(objectA, objectB);
    return 0;
}
```

When a class is made a friend class, all the member functions of that class becomes friend functions. In the following program, all member functions of class B will be friend functions of class A. Thus, any member function of class B can access the private and protected data of class A. But, member functions of class A cannot access the data of class B.

```
class B;
class A
{
    // class B is a friend class of class A
    friend class B;
    … … …
}

class B
{
    … … …
}
```

3.1.8 Inline Functions

The C++ **inline** function is a powerful concept that is commonly used with classes. If a function is inline, the compiler (at compile time) places a copy of the code of that function at each point where the function is called. Therefore, any change to an inline function would require all clients of the function to be recompiled; otherwise they would continue with old functionality.

To inline a function, place the keyword **inline** before the function name and define the function before any calls are made to the function. The compiler can ignore the inline qualifier in event that the function definition is more than one line.

The following example illustrates the use of inline functions:

```
#include <iostream>

using namespace std;

inline int Max(int x, int y)
{
   return (x > y)? x : y; //conditional operator used here
}

// Main function for the program
int main( )
{
   cout << "Max (20,10): " << Max(20,10) << endl;
   cout << "Max (0,200): " << Max(0,200) << endl;
   cout << "Max (100,1010): " << Max(100,1010) << endl;
   return 0;
}
```

3.1.9 Pointer to Class

A pointer to a C++ class works in exactly the same way as a pointer to a structure. In order to access the members (variables and functions) of a pointer to a class you use the member access operator **->**; just as you do with pointers to structures. As with all pointers, you must also initialize the pointer before using it.

3.1.10 'this' Class Pointer

Every object in C++ has access to its own address through an important pointer called **this** pointer. The **this** pointer is an implicit parameter to all member functions. Therefore, inside a member function, this may be used to refer to the invoking object, and access other member functions using the arrow -> operator (as 'this' is a pointer to the object), e.g.:

```
return this->Volume();
```

called from a member function, where `Volume()` is another member function of the same object.

Static functions do not have a 'this' pointer, as they don't belong to a particular instance of a class. Friend functions also do not have a **this** pointer, because friends are not members of a class.

In C++, ‘this’ has two main uses:

1. To pass `*this` or `this` as a parameter to other, non-class methods.

```
void do_something_to_a_foo(Foo *foo_instance);

void Foo::DoSomething()
{
    do_something_to_a_foo(this);
}
```

2. To allow you to remove ambiguities between member variables and function parameters. This is common in constructors.

```
MessageBox::MessageBox(const string& message)
{
  this->message = message;
}
```

3.2 Function and Operator Overloading

3.2.1 Function Overloading

Object-oriented C++ allows you to specify more than one definition for a **class member function** name or an **operator** in the same scope, which are called **function overloading** and **operator overloading** respectively. The overloaded operator function can only take two objects as input.

An overloaded declaration is a declaration that had been declared with the same name as a previously declared declaration in the same scope, except that both declarations have different arguments and different definition. You cannot overload function declarations that differ only by return type.

When you call an overloaded **member function** or **operator**, the compiler determines the most appropriate definition to use by comparing the argument types you used to call the function or operator with the parameter types specified in the declarations. The process of selecting the most appropriate overloaded function or operator is called **overload resolution**.

```
using namespace std;

class printData
{
   public:
      void print(int i) {
        cout << "Printing int: " << i << endl;
      }
      void print(double  f) {
        cout << "Printing float: " << f << endl;
      }
      void print(char* c) {
        cout << "Printing character array: " << c << endl;
      }
};

int main(void)
{
```

```
   printData pd;

   pd.print(5); // Call print to print integer
   pd.print(500.263); // Call print to print float
   pd.print("Hello C++"); // Call print to print character array

  return 0;
}
```

3.2.2 Operator Overloading

You can redefine or overload most of the built-in operators available in C++. Overloaded operators are functions with special names: the keyword operator followed by the symbol for the operator being defined. As opposed to regular overloaded functions, operator overload functions can be member or non-member functions. Operator overloading allows you to redefine the way operator works for user-defined types only (objects, structures). It cannot be used for built-in types (int, float, char etc.).

Like any other function, an overloaded operator has a return type and a parameter list. When defined as a member function of a class, the syntax is as follows (for example):

```
// Overload + operator to add two Box objects.
Box operator+(const Box& b)
{
    Box box;
    box.length = length + b.length;
    box.breadth = breadth + b.breadth;
    box.height = height + b.height;
    return box;
}
```

This declares an addition operator that can be used to add two Box objects and return the final Box object. In case we define the above function as a non-member function (outside of a class), then we would have to pass two arguments for each operand as follows:

```
Box operator+(const Box&, const Box&);
```

3.3 Polymorphism

In C++, polymorphism is used to overload a function defined in the base class, by a function of the same name and parameters (although the parameters can be different) in the derived class. This is implemented using the **virtual** keyword. Depending on where the function is called from (base or a derived class), the appropriate function (located in the correct class; base vs derived) will be selected. This is referred to as **dynamic linkage** (function call linked to different definition based on where it is called from).

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

class Shape {

   protected:
      int width, height;
```

```
   public:

      Shape(int a=0, int b=0)
      {
         width = a;
         height = b;
      }

      virtual int area()
      {
         cout << "Parent class area :" <<endl;
         return 0;
      }
};

class Rectangle: public Shape{
   public:
      Rectangle(int a=0, int b=0):Shape(a, b) { }
      int area ()
      {
         cout << "Rectangle class area :" <<endl;
         return (width * height);
      }
};

class Triangle: public Shape{
   public:
      Triangle(int a=0, int b=0):Shape(a, b) { }
      int area ()
      {
         cout << "Triangle class area :" <<endl;
         return (width * height / 2);
      }
};
// Main function for the program
int main()
{
   Rectangle rec(10,7);
   Triangle tri(10,5);

   //calls function in rec object
   rec.area();
   //calls function in tri object
   tri.area();

   return 0;
}
```

3.4 Abstract Classes and Interfaces

C++ interfaces are implemented using **abstract classes**. A class is made abstract by declaring at least one of its functions as a **pure virtual function**. In order to define a pure virtual function, i.e. a virtual function in the base class that has no body, we can use the following syntax:

```
virtual int area() = 0;
```

- If a class contains variables and virtual functions with definitions, along with one or more pure virtual functions, it is an **abstract class**. Abstract classes cannot be instantiated.

```
class AB {
public:
    virtual void f() = 0; //pure virtual function
};
```

- An **interface** is a class with no state variables and only pure virtual functions. Interfaces classes cannot be instantiated. An interface describes the behavior or capabilities of a class without committing to a particular implementation of that class.

The main purpose of abstract classes or interfaces is to provide an appropriate base class from which other classes can inherit:

- A class inheriting from an interface needs to provide a definition for all pure virtual functions.
- A class inheriting from an abstract class doesn't need to re-define any of the virtual functions in the abstract class, but needs to define the pure virtual functions.
- A class can inherit multiple interfaces, but cannot inherit multiple abstract classes.
- Classes that are not abstract or interfaces (i.e. classes that can be used to instantiate objects) are referred to as **concrete classes**.

Depending on the requirements either an abstract class or an interface can be used:

- If you anticipate creating multiple versions of your class, each with slightly different functionality, create an abstract class. Abstract classes provide a simple and easy way to version your classes. The base class can be easily changed (e.g. by adding variables or functions) without breaking the derived classes (which will also inherit the changes).
- On the other hand, once an interface is defined, it cannot be changed without breaking the functionality of the derived classes, as the derived classes must provide a definition for each pure virtual method in the base class. Therefore, changing an argument or function name in the base class would require a corresponding change in the derived class function.
- If the functionality you are creating will be useful across a wide range of disparate objects, use an interface. Abstract classes should be used primarily for objects that are closely related, whereas interfaces are best suited for providing common functionality to unrelated classes. The unrelated classes can then inherit the bundle of function declarations, and can define each function according to the functionality required in each derived class.

Example of an abstract class:

```cpp
#include <iostream>

using namespace std;

// Base class
class Shape {
   public:
      // pure virtual function providing interface framework.
      virtual int getArea() = 0;
      void setWidth(int w) {
         width = w;
      }

      void setHeight(int h) {
         height = h;
      }

   protected:
      int width;
      int height;
};

// Derived classes
class Rectangle: public Shape {
   public:
      int getArea() {
         return (width * height);
      }
};

class Triangle: public Shape {
   public:
      int getArea() {
         return (width * height)/2;
      }
};

int main(void) {
   Rectangle Rect;
   Triangle  Tri;

   Rect.setWidth(5);
   Rect.setHeight(7);

   // Print the area of the object.
   cout << "Total Rectangle area: " << Rect.getArea() << endl;

   Tri.setWidth(5);
   Tri.setHeight(7);

   // Print the area of the object.
   cout << "Total Triangle area: " << Tri.getArea() << endl;

   return 0;
}
```

3.5 Exception Handling

- **Try/Catch** – try statements and catch exception if it occurs:

```
try
{ // protected code
}
catch(ExceptionType e) //can specify argument to catch any exception type
{ // code to handle ExceptionType exception
}
```

- **Throw** – throw a custom exception anywhere in the code.
- C++ provides an extended library of exceptions, located in the <exception> header file. These provide a wide range of exceptions, each depending on the situation (e.g. bad memory allocation when using the new keyword, or invalid input arguments to a function).
- In C++, it is possible to define your own exceptions by inheriting and overriding the **exception** class functionality.

3.6 Templates

A template is a feature in C++ that allows functions and classes to operate with generic types. This allows a function or class to work on many different data types without being rewritten for each one.

A **function template** behaves like a function except that the template can have arguments of many different types (see example). In other words, a function template represents a family of functions. The format for declaring function templates with type parameters is:

```
template <class identifier> function_declaration; //identifier can be any name (e.g. T)
template <typename identifier> function_declaration;
```

Both expressions have the same meaning and behave in exactly the same way. The latter form was introduced to avoid confusion, since a type parameter need not be a class (it can also be a basic type such as `int` or `double`). For example:

```
template <typename myType>
myType max(myType a, myType b) {
    return a > b ? a : b;
}
```

A template does not produce smaller object code (i.e. compiled code), compared to writing separate functions for all the different data types used in a specific program.

A **class template** can be used to define the member functions of a class, based on the type of data type used when creating an instance of the class. The following example illustrates the use of class templates:

```
template <typename myType>
class Calc
{
  public:
    myType multiply(myType x, myType y);
    myType add(myType x, myType y);
};

template <typename myType>
myType Calc<myType>::multiply(myType x,myType y)
{
  return x*y;
}

template <typename myType>
myType Calc<myType>::add(myType x, myType y)
{
  return x+y;
}
main()
{
// Instantiating class
Calc <double> a_calc_class;
}
```

3.7 Preprocessor Directives

Preprocessor directives give instruction to the compiler to preprocess the information before the actual compilation starts. All preprocessor directives begin with #, and only white-space characters may appear before a preprocessor directive on a line. Preprocessor directives are not C++ statements, so they do not end in a semicolon (;). There are a number of preprocessor directives supported by C++, including **#include**, **#define**, **#if**, **#else**, **#line**:

- **#include**: used to include header files in source files.
- **#define**: used to create symbolic constants, called **macros**. The general form is:

  ```
  #define macro-name replacement-text //(e.g. #define PI 3.14159)
  ```

  #define can also be used to define **function macros**, as follows:

  ```
  #define MIN(a,b) (((a)<(b)) ? a : b)
  ```

- **Conditional Compilation**: directives that can be used to selectively compile portions of the source code. Directives include **#ifdef** (if defined), **#ifndef** (if not defined) and **#if**, **#elif, #else**, **#endif**. These directives operate only on symbolic constants (i.e. #define).

  ```
  #if defined(CREDIT)
      credit();
  #elif defined(DEBIT)
      debit();
  #else
      printerror();
  #endif
  ```

- **#** and **##** preprocessor operators:
    - The **#** operator causes a replacement-text field of a **#define** to be converted to a string surrounded by quotes:

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

#define MKSTR(x) #x //MKSTR is the custom function name, #x is the body

int main ()
{
    cout << MKSTR(HELLO C++) << endl;

    return 0;
}
```

    - The **##** operator can be used to concatenate an expression (not strings, the expression itself, which the compiler interprets):

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

#define concat(a, b) a ## b
int main()
{
   int xy = 100;

   cout << concat(x, y);
   return 0;
}
```

- C++ also has a number of built-in macros, such as __LINE__, __FILE__, __DATE_ and __TIME__, which return the line number of the program, the file name, the system date and the system time respectively.

3.8 Other Concepts in C++

Other concepts in C++ include:

- **Signals**: interrupts delivered to a process by the operating system, which can terminate a program prematurely. You can generate interrupts by pressing Ctrl+C on a UNIX, LINUX, Mac OS X or Windows system. There are signals which cannot be caught by the program, but there is also a list of signals which you can catch in your program and can take appropriate actions based on the signal. These signals are defined in the C++ header file <csignal>.
- **Multithreading**: simultaneous execution of different parts of a program. Each part of the program is referred to as a **thread**.
- **Web Programming**: exchange of information between a custom script and a web server, governed by a set of standards called **Common Gateway Interface** (**CGI**).