



البرمجة المهيكلة

الفصل الثاني

Introduction to C++

Introduction to C++ (Structure of a program)

The most important thing to do when learning C++ is to focus on concepts and not get lost in language-technical details. The purpose of learning a programming language is to become a better programmer; that is, to become more effective at designing and implementing new systems and at maintaining old ones. For this, an appreciation of programming and design techniques is far more important than an understanding of details; that understanding comes with time and practice.

C++ is an object oriented programming language. It was developed by Bjarne Stroustrup at AT&T Bell Laboratories in Murray Hill, New Jersey, USA, in the early 1980's. Stroustrup, an admirer of Simula67 and a strong supporter of C, wanted to combine the best of both the languages and create a more powerful language that could support object-oriented programming features and still retain the power and elegance of C. The result was C++.

A programming language serves two related purposes: it provides a vehicle for the programmer to specify actions to be executed, and it provides a set of concepts for the programmer to use when thinking about what can be done. The first purpose ideally requires a language that is “close to the machine” so that all important aspects of a machine are handled simply and efficiently in a way that is reasonably obvious to the programmer. The second purpose ideally requires a language that is “close to the problem to be solved” so that the concepts of a solution can be expressed directly and concisely. The facilities added to C to create C++ were primarily designed keeping these purposes in mind.

C++ is regarded as a middle-level language, as it comprises a combination of both high-level and low-level language features. C++ was developed as an enhancement to the C language and originally named C with classes.

The main source of inspiration for C++ was C and hence it was initially called C with classes. C is retained as a subset and also C's emphasis on facilities that are low-level enough to cope with the most demanding systems programming tasks. The other main source of inspiration was Simula67; the class concept (with

derived classes and virtual functions) was borrowed from it. C++'s facility for overloading operators and the freedom to place a declaration wherever a statement can occur resembles Algol68. Templates were partly inspired by Ada's generics (both their strengths and weaknesses) and partly by Clu's parameterized modules. Similarly, the C++ exception handling mechanism was inspired partly by Ada, Clu, and ML.

The name C++ (pronounced "see plus plus") was coined by Rick Mascitti in 1983. The name signifies the evolutionary nature of the changes from C; "++" is the C increment operator. The slightly shorter name "C+" is a syntax error; it has also been used as the name of an unrelated language. Connoisseurs of C semantics find C++ inferior to ++C. The language is also not called D, because it is an extension of C, and it does not attempt to remedy problems by removing features.

After years of development, the C++ programming language standard was ratified in 1998 as ISO/IEC 14882:1998. That standard is still current, but is amended by the 2003 technical corrigendum, ISO/IEC 14882:2003.

As one of the most popular programming languages ever created, C++ is used by hundreds of thousands of programmers in essentially every application domain. Some of its application domains include systems software, application software, device drivers, embedded software, high-performance server and client applications, and entertainment software such as video games. C++ has greatly influenced many other popular programming languages, most notable C# and Java. C++ is also used for hardware design, where design is initially described in C++. Below a few of its applications are mentioned elaborately:

Early applications tended to have a strong systems programming flavor. For example, several *major operating systems* have been written in C++ and many more have key parts done in C++.

C++ provides uncompromising low-level efficiency essential for device drivers and other software that rely on direct manipulation of hardware under real-time

constraints. In such code, predictability of performance is at least as important as raw speed.

Most applications have sections of code that are critical for acceptable performance. For most code, maintainability, ease of extension, and ease of testing is key. C++'s support for these concerns has led to its widespread use where reliability is a must and in areas where requirements change significantly over

time. Examples are banking, trading, insurance, telecommunications, and military applications.

Graphics and user interfaces are areas in which C++ is heavily used. Anyone who has used either an Apple Macintosh or a PC running Windows has indirectly used C++ because the primary user interfaces of these systems are C++ programs. In addition, some of the most popular libraries supporting X for UNIX are written in C++. Thus, C++ is a common choice for the vast number of applications in which the user interface is a major part.

A SIMPLE C++ PROGRAM

Before looking at how to write C++ programs consider the following simple example program

```
// my first program in C++  
  
/* my hello world program in C++ with  
   More comments */  
  
#include <iostream>  
using namespace std;  
  
int main ()  
{  
    cout << "Hello World!";  
    return 0;  
}
```

The previous program is the typical program that programmer apprentices write for the first time, and its result is the printing on screen of the "Hello World!"

sentence. It is one of the simplest programs that can be written in C++, but it already contains the fundamental components that every C++ program has. We are going to look line by line at the code we have just written:

```
Or      // my first program in C++
        /* my hello world program in C++ with
           More comments*/
```

This is a comment line and comment block. All lines beginning with two slash signs (//) will represent comment line. Also, all lines beginning with slash followed by star and ended up with star followed by slash will represent comment block (/* */). Both comment line and comment block considered comments and do not have any effect on the behaviour of the program. The programmer can use them to include short explanations or observations within the source code itself. In this case, the line is a brief description of what our program is.

```
#include <iostream>
```

Lines beginning with a hash sign (#) are directives for the preprocessor. They are not regular code lines with expressions but indications for the compiler's preprocessor. In this case the directive `#include<iostream>` tells the preprocessor to include the *iostream* standard file. This specific file (*iostream*) includes the declarations of the basic standard input-output library in C++, and it is included because its functionality is going to be used later in the program.

```
using namespace std;
```

All the elements of the standard C++ library are declared within what is called a namespace, the namespace with the name *std*. So in order to access its functionality we declare with this expression that we will be using these entities. This line is very frequent in C++ programs that use the standard library.

```
int main ()
```

This line corresponds to the beginning of the definition of the *main* function. The *main* function is the point by where all C++ programs start their execution, independently of its location within the source code. It does not matter whether there are other functions with other names defined before or after it - the instructions contained within this function's definition will always be the first ones to be executed in any C++ program. For that same reason, it is essential that all C++ programs have a main function. The word *main* is followed in the code by a pair of parentheses (`()`). That is because it is a function declaration. In C++, what differentiate a function declaration from other expressions are the parentheses that follow its name. Optionally, these parentheses may enclose a list of parameters within them. Right after these parentheses we can find the body of the main function enclosed in braces (`{ }`). What is contained within these braces is what the function does when it is executed.

```
cout << "Hello World!";
```

This line is a C++ statement. A statement is a simple or compound expression that can actually produce some effect. In fact, this statement performs the only action that generates a visible effect in our first program.

cout: represents the **standard output** stream in C++, and the meaning of the entire statement is to insert a sequence of characters (in this case the Hello World sequence of characters) into the standard output stream (which usually is the screen). *cout* is declared in the *iostream* standard file within the `std` namespace, so that's why we needed to include that specific file and to declare that we were going to use this specific namespace earlier in our code. Notice that the statement ends with a semicolon character (`;`). This character is used to mark the end of the statement and in fact it must be included at the end of all expression statements in all C++ programs (one of the most common syntax errors is indeed to forget to include some semicolon after a statement).

```
return 0;
```

The *return* statement causes the main function to finish. *return* may be followed by a return code (in our example is followed by the return code 0). A return code of 0 for the main function is generally interpreted as the program worked as expected without any errors during its execution. This is the most usual way to end a C++ console program.

You may have noticed that not all the lines of this program perform actions when the code is executed. There were lines containing only comments (those beginning by //). There were lines with directives for the compiler's pre-processor (those beginning by #). Then there were lines that began the declaration of a function (in this case, the main function) and, finally lines with statements (like the insertion into *cout*), which were all included within the block delimited by the braces ({}) of the main function.

The program has been structured in different lines in order to be more readable, but in C++, we do not have strict rules on how to separate instructions in different lines. For example, instead of

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

We could have written:

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

All in just one line and this would have had exactly the same meaning as the previous code. In C++, the separation between statements is specified with an ending semicolon (;) at the end of each one, so the separation in different code lines does not matter at all for this purpose. We can write many statements per line or write a single statement that takes many code lines. The division of code in different lines serves only to make it more legible and schematic for the humans that may read it.

At this stage let us consider the general format of a C++ program:

```
//Introductory comment
//file name, programmer, when written or modified
/* what program does.....
.....
.....
..... */

#include <iostream>

using namespace std;

int main()
{
    constant declarations
    variable declarations
    executable statements
}
```

Note that it makes complex programs much easier to interpret if, as above, closing braces (}) are aligned with the corresponding opening brace ({). However other conventions are used for the layout of braces in textbooks and other C++ programmers' programs. Also additional spaces, new lines etc. can also be used to make programs more readable. The important thing is to adopt one of the standard conventions and stick to it consistently.

The format given above is the general format of a C++ program. Although this structure will also include the class declaration and member functions definitions but we will come elaborate on that part later.

EX1: Write a C++ program that prints on screen the following sentences:

```
{****Shatt Al-arab University College****}
```

```
{* Computer Science Department*} {**First Year**}
```

```
A C++ program that prints on screen the following sentences:
    {****Shatt Al-arab University College****}
    {* Computer Science Department *}
    {**First Year**}
#include <iostream>
using namespace std;
int main()
{
    cout << "{****Shatt Al-arab University College****}" << endl;
    cout << "    {* Computer Science Department *}"
    <<endl;
    cout << "                {**First Year**}" << endl;
    return 0;
}
```

Cin: is the **standard input** device is usually the keyboard. Handling the standard input in C++ is done by applying the overloaded operator of extraction (>>) on the cin stream. The operator must be followed by the variable that will store the data that is going to be extracted from the stream. For example:

```
int age
cin >> age;
```

The first statement declared a **variable of type integer (int)** called age, and the second one waits for an input form cin (the keyboard) in order to store it in this integer variable.

You can also use **cin** to request more than one datum input from the user:

```
cin >> a >> b;
```

is equivalent to :

```
cin >> a;  
cin >> b;
```

EX2: Write a C++ program that enter any integer value and find its double, the screen output should be like:

```
Please enter an integer value: 10  
The value you entered is 10 and its double is 20
```

```
/* A C++ program that enter any integer value  
and find its double, the screen output should be like:  
Please enter an integer value: 10  
The value you entered is 10 and its double is 20  
*/  
  
#include <iostream>  
using namespace std;  
int main ( )  
{  
    int i;  
    cout << "Please enter an integer value: ";  
    cin >> i;  
    cout << "The value you entered is " << i;  
    cout << " and its double is " << i*2;  
    return 0;  
}
```

1. Variables

A variable is a symbolic name for a memory location in which data can be stored and subsequently recalled. Variables are used for holding data values so that they can be utilized in various computations in a program. All variables have two important attributes:

- A type which is established when the variable is defined (e.g., integer, real, character). Once defined, the type of a C++ variable cannot be changed.
- A value which can be changed by assigning a new value to the variable. The kind of values a variable can assume depends on its type. For example, an integer variable can only take integer values (e.g., 2, 100, -12).
- A variable is used for the quantities which are manipulated by a computer program. For example a program that reads a series of numbers and sums them will have to have a variable to represent each number as it is entered and a variable to represent the sum of the numbers.

Let us illustrate the uses of some simple variable.

```
int workDays;  
float workHours, payRate, weeklyPay;
```

Suppose the above given part of the program aims to calculate the weekly pay using three variables `workDays`, `workHours`, and `payRate`.

In order to distinguish between different variables, they must be given **identifiers**. Each variable needs an identifier that distinguishes it from the others, for example, in the previous code the variable identifiers were `weeklyPay`, `workDays`, `workHours` and `payRate`, but we could have called the variables any names we wanted to invent, as long as they were valid identifiers.

2. Identifiers

Identifiers are names given to variables which distinguish them from all other variables. The rules of C++ for valid identifiers state that:

An identifier must:

start with a letter

consist only of letters, the digits 0-9, or the underscore symbol

–

Not be a **reserved word**.

Reserved words are otherwise valid identifiers that have special significance to C++. For the purposes of C++ identifiers, the underscore symbol, `_`, is considered to be a letter. Its use as the first character in an identifier is not recommended though, because many library functions in C++ use such identifiers. Similarly, the use of two consecutive underscore symbols, `__`, is forbidden.

The following are valid identifiers:

```
Length,   days_in_year,   DataSet1,   Profit95,  
_Pressure, first_one,   first_1 although using  
_Pressure is not recommended.
```

The following are invalid:

```
days-in-year ldata int first.val throw
```

Identifiers should be chosen to reflect the significance of the variable in the program being written. Although it may be easier to type a program consisting of single character identifiers, modifying or correcting such a program becomes more and more difficult. The minor typing effort of using *meaningful* identifiers will repay itself many fold in the avoidance of simple programming errors when the program is modified.

At this stage it is worth noting that C++ is **case-sensitive**. That is lower-case letters are treated as distinct from upper-case letters. Thus the word days in a program is quite different from the word Days or the word DAYS.

3. Reserved Words

The **syntax rules** (or grammar) of C++ define certain symbols to have a unique meaning within a C++ program. These symbols, the **reserved words**, must not be used for any other purposes. Reserved words are otherwise valid identifiers that have special significance to C++. All reserved words are in lower-case letters. The table below lists the reserved words of C++.

asm	auto	bool	break	case
catch	char	class	const_cast	continue
default	delete	do	double	else
enum	dynamic_cast	extern	false	float
for	union	unsigned	using	friend
goto	if	inline	int	long
mutable	virtual	namespace	new	operator
private	protected	public	register	void
reinterpret_cast	return	short	signed	sizeof
static	static_cast	volatile	struct	switch
template	this	throw	true	try
typedef	typeid	unsigned	wchar_t	while

Some of these reserved words may not be treated as reserved by older compilers. However you would do well to avoid their use. Other compilers may add their own reserved words. Typical are those used by Borland compilers for the computer, which add `near`, `far`, `huge`, `cdecl`, and `pascal`.

Notice that `main` is *not* a reserved word. However, this is a fairly technical distinction, and for practical purposes you are advised to treat `main`, `cin`, and `cout` as if they were reserved as well.

4. Data Types

Every name (identifier) in a C++ program has a type associated with it. This type determines what operations can be applied to the name (that is, to the entity referred to by the name) and how such operations are interpreted. For example, the declarations

```
float x;           //x is a floating-point variable
int y = 7;        /*y is an integer variable with
the initial value 7*/
float f(int);     /*f is a function taking an
argument of type int and returning a floating-
point number */
```

would make the example meaningful. Because `y` is declared to be an `int`, it can be assigned to, used in arithmetic expressions, etc. On the other hand, `f` is declared to be a function that takes an `int` as its argument, so it can be called given a suitable argument.

Data types in Standard C++ are classified as shown in the diagram below.

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

The ***Boolean***, ***character***, and ***integer*** types are collectively called integral types. The ***integer*** and ***floating-point*** types are collectively called ***arithmetic*** types. ***Enumeration*** and structured types are called ***user-defined*** types because they must be defined by users rather than being available for use without previous declaration, the way fundamental types are. In contrast, other types are called ***built-in*** types.

The ***integer*** and ***floating-point*** types are provided in a variety of sizes to give the programmer a choice of the amount of storage consumed, the precision, and the range available for computations. The assumption is that a computer provides bytes for holding characters and words, for holding and computing - integer values, some entity most suitable for floating-point computation, and addresses for referring to those entities. The C++ fundamental types together with ***pointers*** and ***arrays*** present these machine-level notions to the programmer in a reasonably implementation independent manner.

For most applications, one could simply use *bool* for logical values, *char* for characters, *int* for integer values, and *double* for floating-point values. The remaining fundamental types are variations for optimizations and special needs that are best ignored until such needs arise. They must be known, however, to read old C and C++ code.

4.1. Booleans

A Boolean, *bool*, can have one of the two values *true* or *false*. A Boolean is used to express the results of logical operations. For example:

```
bool b1 = a == b; // = is assignment, == is Equality
// ...
```

If *a* and *b* have the same value, *b1* becomes *true*; otherwise, *b1* becomes *false*. A common use of *bool* is as the type of the result of a function that tests some condition (a predicate). For example:

```
bool b = 7; // bool(7) is true, so b becomes true
int i = true; // int(true) is 1, so i becomes 1
```

In arithmetic and logical expressions, *bools* are converted to *ints*; integer arithmetic and logical operations are performed on the converted values. If the result is converted back to *bool*, a 0 is converted to *false* and a nonzero value is converted to *true*.

EX1: What is the screen output after executing the following C++ program?

```
#include <iostream>
using namespace std;
int main()
{
    bool a = true;
    bool b = true;
    bool x;
```



```
bool y;  
x=a+b;  
y=a-b;  
cout<<"The value of x is "<< x << endl;  
cout<<"The value of y is "<< y << endl;  
}
```

4.2 Character Types:

A variable of type `char` can hold a character of the implementation's character set. For example:

```
char ch = 'a';
```

Almost universally, a `char` has 8 bits so that it can hold one of 256 different values. Typically, the character set is a variant of ISO-646, for example ASCII, thus providing the characters appearing on your keyboard. Each character constant has an integer value. For example, the value of 'b' is 98 in the ASCII character set. Here is a small program that will tell you the integer value of any character you care to input:

EX2: What is the screen output after executing the following C++ program?

```
#include <iostream>  
using namespace std;  
int main()  
{  
    char c;  
    cin>> c;  
    cout<< "The value of" << c << "is" << int(c) << endl;  
}
```

The notation `int(c)` gives the integer value for a character `c`. The possibility of converting a `char` to an integer raises the question: is a `char` signed or unsigned? The 256 values represented by an 8-bit byte can be interpreted as the values 0 to 255 or as the values -127 to 127. Unfortunately, which choice is made for a plain `char` is implementation-defined. C++ provides two types for which the answer is definite; signed `char`, which can hold at least the values -127 to 127, and unsigned `char`, which can hold at least the values 0 to 255. Fortunately, the difference

matters only for values outside the 0 to 127 range, and the most common characters are within that range.

4.3 Integer Types:

Like char, each integer type comes in three forms: “plain” int, signed int, and unsigned int. In addition, integers come in three sizes: short int, “plain” int, and long int. A long int can be referred to as plain long. Similarly, short is a synonym for short int, unsigned for unsigned int, and signed for signed int. The unsigned integer types are ideal for uses that treat storage as a bit array. Using an unsigned instead of an int to gain one more bit to represent positive integers is almost never a good idea. Attempts to ensure that some values are positive by declaring variables unsigned will typically be defeated by the implicit conversion rules. Unlike plain chars, plain ints are always signed. The signed int types are simply more explicit synonyms for their plain int counterparts.

```
int x;  
long int y;  
short int z;
```

4.4 Floating-Point Types:

The floating-point types represent floating-point numbers. Like integers, floating-point types come in three sizes: *float* (single-precision), *double* (double-precision), and *long double* (extended-precision).

The exact meaning of single-, double-, and extended-precision is implementation-defined. Choosing the right precision for a problem where the choice matters requires significant understanding of floating-point computation. If you don’t have that understanding, get advice, take the time to learn, or use *double* and hope for the best.

```
float a;  
float b;
```

```
double x;  
long double y;  
short double z;
```

Example about all data type

```
#include <iostream>  
using namespace std;  
  
// Global Variable declaration:  
int a, b;  
int c;  
float f;  
  
int main () {  
    // Local Variable definition:  
    int a, b;  
    int c;  
    float f;  
  
    // actual initialization  
    a = 10;  
    b = 20;  
    c = a + b;  
  
    cout << c << endl ;  
  
    f = 70.0/3.0;  
    cout << f << endl ;  
  
    return 0;  
}
```

5. Declaration of variables

In C++ (as in many other programming languages) all the variables that a program is going to use must be **declared** prior to use. Declaration of a variable serves two purposes:

It associates a **type** and an identifier (or name) with the variable. The type allows the compiler to interpret statements correctly. For example in the CPU the instruction to add two integer values together is different from the instruction to add two floating-point values together. Hence the compiler must know the type of the variables so it can generate the correct add instruction.

It allows the compiler to decide how much storage space to allocate for storage of the value associated with the identifier and to assign an address for each variable which can be used in code generation.

A typical set of variable declarations that might appear at the beginning of a program could be as follows:

```
int i, j, count;
float sum, product;
char ch;
bool passed_exam;
```

which declares integer variables `i`, `j` and `count`, real variables `sum` and `product`, a character variable `ch`, and a boolean variable `pass_exam`.

A variable declaration has the form:

type identifier-list;

type specifies the type of the variables being declared. The **identifier-list** is a list of the identifiers of the variables being declared, separated by commas.

Variables may be initialised at the time of declaration by assigning a value to them as in the following example:

```
int i, j, count = 0;
float sum = 0.0, product;
char ch = '7';
bool passed_exam = false;
```

which assigns the value 0 to the integer variable `count` and the value 0.0 to the real variable `sum`. The character variable `ch` is initialised with the character 7. `i`, `j`, and `product` have no initial value specified, so the program should make *no* assumption about their contents.

To see what variable declarations look like in action within a program, we are going to see the following C++ code.

EX3: What is the screen output after executing the following C++ program?

```
// operating with variables

#include <iostream>
using namespace std;

int main ()
{
    declaring variables:
    int a, b;
    int result;
    a = 5;
    b = 2;
    a = a + 1;
    result = a - b;
    cout << result;
    return 0;
}
```

Do not worry if something else than the variable declarations themselves looks a bit strange to you. You will see the rest in detail in coming sections.

6. Initialization of variables

When declaring a regular local variable, its value is by default undetermined. But you may want a variable to store a concrete value at the same moment that it is declared. In order to do that, you can initialize the variable. There are two ways to do this in C++:

The first one, known as c-like, is done by appending an equal sign followed by the value to which the variable will be initialized:

```
type identifier = initial_value ;
```

For example, if we want to declare an int variable called a initialized with a value of 0 at the moment in which it is declared, we could write:

```
int a = 0;
```

The other way to initialize variables, known as constructor initialization, is done by enclosing the initial value between parentheses (()):

```
type identifier (initial_value) ;
```

For example:

```
int a (0);
```

Both ways of initializing variables are valid and equivalent in C++.

EX4: What is the screen output after executing the following C++ program?

```
// initialization of variables

#include <iostream>
using namespace std;

int main ()
{
    int a=5; // initial value = 5
    int b(2); // initial value = 2
    int result; // initial value undetermined
    a = a + 3;
    result = a - b;
    cout << result;
    return 0;
}
```

7. Constants

Constants are expressions with a fixed value. C++ has two kinds of constants: literal, and symbolic. C++ has two kinds of constants: literal, and symbolic.

7.1 Literal constants:

Literal constants are literal numbers used to express particular values within the source code of a program. They are constants because you can't change their values. We have already used these previously to give concrete values to variables or to express messages we wanted our programs to print out, for example, when we wrote:

```
double a=5.8; // initial value = 5.8
int b(2); // initial value = 2
char ch = 'H'; // initial char value= H
bool a = true; // initial a value= 1
```

7.2 Symbolic constants:

Symbolic constants can be declared in two different ways: using the *#define* preprocessor directive, and through use of the *const* keyword.

You can define your own names for constants that you use very often without having to resort to memory consuming variables, simply by using the *#define* preprocessor directive. Its format is:

```
#define identifier value
```

For example:

```
#define PI 3.14159
#define NEWLINE '\n'
```

This defines two new constants: *PI* and *NEWLINE*. Once they are defined, you can use them in the rest of the code as if they were any other regular constant, for example:

EX5: What is the screen output after executing the following C++ program?

```
// defined constants: calculate circumference

#include <iostream>
using namespace std;
#define PI 3.14159
#define NEWLINE '\n'

int main ()
{
    double r=5.0;
    double circumf; // radius
    circumf = 2 * PI * r;
    cout << circumf;
    cout << NEWLINE;
    return 0;
}
```

OUTPUT: 31.4159

In fact the only thing that the compiler preprocessor does when it encounters *#define* directives is to literally replace any occurrence of their identifier (in the

previous example, these were PI and NEWLINE) by the code to which they have been defined (3.14159 and '\n' respectively).

There are two major problems with symbolic constants declared using #define. First, because they are resolved by the preprocessor, which replaces the symbolic name with the defined value, #defined symbolic constants do not show up in the debugger. Second, #defined values always have global scope. This means value #defined in one piece of code may have a naming conflict with a value #defined with the same name in another piece of code.

A better way to do symbolic constants is through use of the *const* keyword. Const variables must be assigned a value when declared, and then that value cannot be changed.

```
const double pi = 3.14159
const char newline = '\n'
```

Here, *pi* and *newline* are two typed constants. Although a constant variable might seem like an oxymoron, they can be very useful in helping to document your code. Const variables act exactly like normal variables in every case except that they cannot be assigned to.

EX6: Write a C++ program to calculate the weekly pay?

`weeklyPay = workDays * workHours * payRate;`

```
#include <iostream>
using namespace std;
int main (void)
{
    int workDays;
    float workHours, payRate, weeklyPay;
    workDays = 5;
    workHours = 7.5;
    payRate = 38.55;
    weeklyPay = workDays * workHours * payRate;
    cout << "Weekly Pay = ";
    cout << weeklyPay;
    cout << '\n';
}
```

EX7: Write a C++ program to apply four different types of data and outputs it?

```
#include<iostream>
using namespace std;
int main( )
{
    bool pass=true;
    cout << "bool= "<<pass<<'\n';
    int num=3;
    cout << "Number= "<<num<<'\n';
    char ch='a';
    cout << "Character= "<<ch<<'\n';
    float fa=34.45;
    cout<<"Float number= "<<fa<<'\n';
}
```

EX8: Write a C++ program to input three different types of data and outputs it?

```
#include<iostream>
using namespace std;
int main( )
{
    int n; float f; char c;
    cout << "input integer number: ";
    cin>>n;
    cout<<n<<endl;
    cout << "input float number: ";
    cin>>f;
    cout<<f<<endl;
    cout << "input character: ";
    cin>>c ;
    cout<< c ;
    return 0;
}
```

EX9: Write a C++ program that reads the radius of a circle, then outputs its area?

```
#include<iostream>
using namespace std;
int main( )
{
    float radu, Area_cir;
    const float pi = 3.14;
    cout << "enter the radius of circle: ";
    cin>>radu;
    cout<<endl;
    Area_cir = radu * radu * pi;
    cout << "the area of circle: " << Area_cir;
    cout <<endl;
    return 0;
}
```

8. Operators in C++



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

1. Arithmetic Operators
2. Relational Operators
3. Logical Operators
4. Assignment Operators

8.1. Arithmetic Operators

There are following arithmetic operators supported by C++ language –

Assume variable A holds 10 and variable B holds 20, then –

Operator	Description	Example
+	Adds two operands	A + B will give 30
-	Subtracts second operand from the first	A - B will give -10
*	Multiplies both operands	A * B will give 200
/	Divides numerator by de-numerator	B / A will give 2
%	Modulus Operator and remainder of after an integer division	B % A will give 0
++	Increment operator  , increases integer value by one	A++ will give 11
--	Decrement operator  , decreases integer value by one	A-- will give 9

EX9: Example about Arithmetic Operators

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

int main() {
    int a = 21;
    int b = 10;
    int c ;
    c = a + b;
    cout << "Line 1 - Value of c is :" << c << endl ;
    c = a - b;
    cout << "Line 2 - Value of c is  :" << c << endl ;
    c = a * b;
    cout << "Line 3 - Value of c is :" << c << endl ;
    c = a / b;
    cout << "Line 4 - Value of c is  :" << c << endl ;
    c = a % b;
    cout << "Line 5 - Value of c is  :" << c << endl ;
    c = a++;
    cout << "Line 6 - Value of c is :" << c << endl ;
    c = a--;
    cout << "Line 7 - Value of c is  :" << c << endl ;
    return 0;
}
```

8.2. Relational Operators

There are following relational operators supported by C++ language

Assume variable A holds 10 and variable B holds 20, then –

Operator	Description	Example
==	Checks if the values of two operands are equal or not, if yes then condition becomes true.	(A == B) is not true.
!=	Checks if the values of two operands are equal or not, if values are not equal then condition becomes true.	(A != B) is true.
>	Checks if the value of left operand is greater than the value of right operand, if yes then condition becomes true.	(A > B) is not true.
<	Checks if the value of left operand is less than the value of right operand, if yes then condition becomes true.	(A < B) is true.
>=	Checks if the value of left operand is greater than or equal to the value of right operand, if yes then condition becomes true.	(A >= B) is not true.
<=	Checks if the value of left operand is less than or equal to the value of right operand, if yes then condition becomes true.	(A <= B) is true.

EX10: Example about Relational Operators

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

int main() {
    int a = 21;
    int b = 10;
    int c ;

    if( a == b ) {
        cout << "Line 1 - a is equal to b" << endl ;
    } else {
        cout << "Line 1 - a is not equal to b" << endl ;
    }

    if( a < b ) {
        cout << "Line 2 - a is less than b" << endl ;
    } else {
        cout << "Line 2 - a is not less than b" << endl ;
    }

    if( a > b ) {
        cout << "Line 3 - a is greater than b" << endl ;
    } else {
        cout << "Line 3 - a is not greater than b" << endl
;
    }

    /* Let's change the values of a and b */
    a = 5;
    b = 20;
    if( a <= b ) {
        cout << "Line 4 - a is either less than \ or equal
to b" << endl ;
    }

    if( b >= a ) {
        cout << "Line 5 - b is either greater than \ or
equal to b" << endl ;
    }

    return 0;
}
```

8.3 Logical Operators

There are following logical operators supported by C++ language.

Assume variable A holds 1 and variable B holds 0, then –

Operator	Description	Example
&&	Called Logical AND operator. If both the operands are non-zero, then condition becomes true.	(A && B) is false.
	Called Logical OR Operator. If any of the two operands is non-zero, then condition becomes true.	(A B) is true.
!	Called Logical NOT Operator. Use to reverses the logical state of its operand. If a condition is true, then Logical NOT operator will make false.	!(A && B) is true.

EX11: Example about Logical Operators

```
#include <iostream>
using namespace std;
int main() {
    int a = 5;
    int b = 20;
    int c ;
    if(a && b) {
        cout << "Line 1 - Condition is true"<< endl ;
    }
    if(a || b) {
        cout << "Line 2 - Condition is true"<< endl ;
    }
    /* Let's change the values of a and b */
    a = 0;
    b = 10;
    if(a && b) {
        cout << "Line 3 - Condition is true"<< endl ;
    } else {
        cout << "Line 4 - Condition is not true"<< endl ;
    }
    if(!(a && b)) {
        cout << "Line 5 - Condition is true"<< endl ;
    }
    return 0;
}
```

8.4 Assignment Operators

There are following assignment operators supported by C++ language –

Operator	Description	Example
=	Simple assignment operator, Assigns values from right side operands to left side operand.	$C = A + B$ will assign value of $A + B$ into C
+=	Add AND assignment operator, It adds right operand to the left operand and assign the result to left operand.	$C += A$ is equivalent to $C = C + A$
-=	Subtract AND assignment operator, It subtracts right operand from the left operand and assign the result to left operand.	$C -= A$ is equivalent to $C = C - A$
*=	Multiply AND assignment operator, It multiplies right operand with the left operand and assign the result to left operand.	$C *= A$ is equivalent to $C = C * A$
/=	Divide AND assignment operator, It divides left operand with the right operand and assign the result to left operand.	$C /= A$ is equivalent to $C = C / A$
%=	Modulus AND assignment operator, It takes modulus using two operands and assign the result to left operand.	$C \% = A$ is equivalent to $C = C \% A$

EX12: Example about Assignment Operators

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

int main() {
    int a = 21;
    int c ;

    c = a;
```



```
    cout << "Line 1 - = Operator, Value of c = : " <<c<<
endl ;

    c += a;
    cout << "Line 2 - += Operator, Value of c = : " <<c<<
endl ;

    c -= a;
    cout << "Line 3 - -= Operator, Value of c = : " <<c<<
endl ;

    c *= a;
    cout << "Line 4 - *= Operator, Value of c = : " <<c<<
endl ;

    c /= a;
    cout << "Line 5 - /= Operator, Value of c = : " <<c<<
endl ;

    c = 200;
    c %= a;
    cout << "Line 6 - %= Operator, Value of c = : " <<c<<
endl ;

    return 0;
}
```

9. Operators Precedence in C++

Operator precedence determines the grouping of terms in an expression. This affects how an expression is evaluated. Certain operators have higher precedence than others; for example, the multiplication operator has higher precedence than the addition operator –

For example $x = 7 + 3 * 2$; here, x is assigned 13, not 20 because operator $*$ has higher precedence than $+$, so it first gets multiplied with $3*2$ and then adds into 7.

Here, operators with the highest precedence appear at the top of the table, those with the lowest appear at the bottom. Within an expression, higher precedence operators will be evaluated first.

Category	Operator	Associativity
Postfix	<code>()</code> , <code>[]</code> , <code>-</code> , <code>++</code> , <code>--</code>	Left to right
Multiplicative	<code>*</code> , <code>/</code> , <code>%</code>	Left to right
Additive	<code>+</code> , <code>-</code>	Left to right
Relational	<code><</code> , <code><=</code> , <code>></code> , <code>>=</code>	Left to right
Equality	<code>==</code> , <code>!=</code>	Left to right
Logical AND	<code>&&</code>	Left to right
Logical OR	<code> </code>	Left to right
Assignment	<code>=</code> , <code>+=</code> , <code>-=</code> , <code>*=</code> , <code>/=</code> , <code>%=</code>	Right to left

EX13: Example about Assignment Operators

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

int main() {
    int a = 20;
    int b = 10;
    int c = 15;
    int d = 5;
    int e;

    e = (a + b) * c / d;          // ( 30 * 15 ) / 5
    cout << "Value of (a + b) * c / d is :" << e << endl ;

    e = ((a + b) * c) / d;      // (30 * 15 ) / 5
    cout << "Value of ((a + b) * c) / d is  :" << e <<
endl ;

    e = (a + b) * (c / d);     // (30) * (15/5)
    cout << "Value of (a + b) * (c / d) is  :" << e <<
endl ;

    e = a + (b * c) / d;      // 20 + (150/5)
    cout << "Value of a + (b * c) / d is  :" << e << endl
;

    return 0;
}
```