UNIT-I

C++ is a statically typed, 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++ was developed by Bjarne Stroustrup starting in 1979 at Bell Labs in Murray Hill, New Jersey, as an enhancement to the C language and originally named C with Classes but later it was renamed C++ in 1983.

C++ is a superset of C, and that virtually any legal C program is a legal C++ program.

Note − A programming language is said to use static typing when type checking is performed during compile-time as opposed to run-time.

Object-Oriented Programming

C++ fully supports object-oriented programming, including the four pillars of object-oriented development −

- Encapsulation
- Data hiding
- Inheritance
- Polymorphism

Standard Libraries

Standard C++ consists of three important parts −

- The core language giving all the building blocks including variables, data types and literals, etc.
- The C++ Standard Library giving a rich set of functions manipulating files, strings, etc.
- The Standard Template Library (STL) giving a rich set of methods manipulating data structures, etc.

The ANSI Standard

The ANSI standard is an attempt to ensure that C++ is portable; that code you write for Microsoft's compiler will compile without errors, using a compiler on a Mac, UNIX, a Windows box, or an Alpha.

The ANSI standard has been stable for a while, and all the major C++ compiler manufacturers support the ANSI standard.

Learning C++

The most important thing while learning C++ is to focus on concepts.

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.

C++ supports a variety of programming styles. You can write in the style of Fortran, C, Smalltalk, etc., in any language. Each style can achieve its aims effectively while maintaining runtime and space efficiency.

Use of C++

C++ is used by hundreds of thousands of programmers in essentially every application domain.

C++ is being highly used to write device drivers and other software that rely on direct manipulation of hardware under realtime constraints.

C++ is widely used for teaching and research because it is clean enough for successful teaching of basic concepts.

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 written in C++.

FUNCTIONS:

A function is a group of statements that together perform a task. Every C++ program has at least one function, which is **main()**, and all the most trivial programs can define additional functions.

You can divide up your code into separate functions. How you divide up your code among different functions is up to you, but logically the division usually is such that each function performs a specific task.

A function **declaration** tells the compiler about a function's name, return type, and parameters. A function **definition** provides the actual body of the function.

A function is known with various names like a method or a sub-routine or a procedure etc.

Defining a Function

The general form of a C++ function definition is as follows −

```
return_type function_name( parameter list )
{
  body of the function
}
```

A C++ function definition consists of a function header and a function body. Here are all the parts of a function −

- **Return Type** − A function may return a value. The **return_type** is the data type of the value the function returns. Some functions perform the desired operations without returning a value. In this case, the return_type is the keyword **void**.
- **Function Name** − This is the actual name of the function. The function name and the parameter list together constitute the function signature.
- **Parameters** − A parameter is like a placeholder. When a function is invoked, you pass a value to the parameter. This value is referred to as actual parameter or argument. The parameter list refers to the type, order, and number of the parameters of a function. Parameters are optional; that is, a function may contain no parameters.
- **Function Body** − The function body contains a collection of statements that define what the function does.

Example

Following is the source code for a function called **max()**. This function takes two parameters num1 and num2 and return the biggest of both −

```
// function returning the max between two numbers

int max(int num1, int num2) {
  // local variable declaration
  int result;

  if (num1 > num2)
    result = num1;
  else
    result = num2;

  return result;
}
```

Function Declarations

A function **declaration** tells the compiler about a function name and how to call the function. The actual body of the function can be defined separately.

A function declaration has the following parts −

```
return_type function_name( parameter list );
```

For the above defined function max(), following is the function declaration −

int max(int num1, int num2);

Parameter names are not important in function declaration only their type is required, so following is also valid declaration –

int max(int, int);

Function declaration is required when you define a function in one source file and you call that function in another file. In such case, you should declare the function at the top of the file calling the function.

Calling a Function

While creating a C++ function, you give a definition of what the function has to do. To use a function, you will have to call or invoke that function.

When a program calls a function, program control is transferred to the called function. A called function performs defined task and when it's return statement is executed or when its function-ending closing brace is reached, it returns program control back to the main program.

To call a function, you simply need to pass the required parameters along with function name, and if function returns a value, then you can store returned value. For example –

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

// function declaration
int max(int num1, int num2);

int main () {
  // local variable declaration:
  int a = 100;
  int b = 200;
  int ret;

  // calling a function to get max value.
  ret = max(a, b);
  cout << "Max value is : " << ret << endl;

  return 0;
}

// function returning the max between two numbers
int max(int num1, int num2) {
  // local variable declaration
  int result;
```

```
  if (num1 > num2)
    result = num1;
  else
    result = num2;

  return result;
}
```

I kept max() function along with main() function and compiled the source code. While running final executable, it would produce the following result –

Max value is : 200

Function Arguments

If a function is to use arguments, it must declare variables that accept the values of the arguments. These variables are called the **formal parameters** of the function.

The formal parameters behave like other local variables inside the function and are created upon entry into the function and destroyed upon exit.

While calling a function, there are two ways that arguments can be passed to a function –

Call by value and call by reference in C++

There are two ways to pass value or data to function in C language: call by value and call by reference. Original value is not modified in call by value but it is modified in call by reference.

Call by value in C++

In call by value, **original value is not modified.**

In call by value, value being passed to the function is locally stored by the function parameter in stack memory location. If you change the value of function parameter, it is changed for the current function only. It will not change the value of variable inside the caller method such as main().

```
#include <iostream>
using namespace std;
void change(int data);
```

```
int main()
{
int data = 3;
change(data);
cout << "Value of the data is: " << data<< endl;
return 0;
}
void change(int data)
{
data = 5;
}
```

Output:

```
Value of the data is: 3
```

Call by reference in C++

In call by reference, original value is modified because we pass reference (address).

Here, address of the value is passed in the function, so actual and formal arguments share the same address space. Hence, value changed inside the function, is reflected inside as well as outside the function.

Note: To understand the call by reference, you must have the basic knowledge of pointers.

Let's try to understand the concept of call by reference in C++ language by the example given below:

```
#include<iostream>
using namespace std;
void swap(int *x, int *y)
{
 int swap;
 swap=*x;
```

```
 *x=*y;
 *y=swap;
}
int main()
{
 int x=500, y=100;
 swap(&x, &y);  // passing value to function
 cout<<"Value of x is: "<<x<<endl;
 cout<<"Value of y is: "<<y<<endl;
 return 0;
}
```

Output:

```
Value of x is: 100
Value of y is: 500
```

Difference between call by value and call by reference in C++

No.	Call by value	Call by reference
1	A copy of value is passed to the function	An address of value is passed to the function
2	Changes made inside the function is not reflected on other functions	Changes made inside the function is reflected outside the function also
3	Actual and formal arguments will be created in different memory location	Actual and formal arguments will be created in same memory location

C++ inline function

C++ **inline** function is powerful concept that is commonly used with classes. If a function is inline, the compiler places a copy of the code of that function at each point where the function is called at compile time.

Any change to an inline function could require all clients of the function to be recompiled because compiler would need to replace all the code once again otherwise it will 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 case defined function is more than a line.

A function definition in a class definition is an inline function definition, even without the use of the **inline** specifier.

Following is an example, which makes use of inline function to return max of two numbers –

```
#include <iostream>
 using namespace std;
inline int Max(int x, int y)
 {
  return (x > y)? x : y;
}

// 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;
}
```

When the above code is compiled and executed, it produces the following result –

```
Max (20,10): 20
Max (0,200): 200
Max (100,1010): 1010
```

Inline

- It is a function in C++.
- It is parsed by the compiler.
- It can be defined inside or outside the class.
- It evaluates the argument only once.
- The compiler may not convert all functions to 'inline' function and expand them all.
- The short functions that are defined inside the class are automatically made as inline functions.
- An inline function inside a class can access the data members of the class.
- Inline function can be terminated using curly brackets.
- It is easy to debug.

- This is because error checking is done during compilation.
- It binds all statements in the body of the function.

Example

```
inline return_type funct_name ( parameters ) {
  . . .
}
```

Macro

- It is expanded by the preprocessor.
- It is defined at the beginning of the program.
- It evaluates the argument every time it is used inside the code.
- They always need to be/are expanded.
- They need to be defined specifically.
- They will never become members of class.
- They can't access data members of the class.
- Definition of macro ends with the new line.
- It is difficult to debug macros since error checking doesn't happen during compile time.
- It encounters binding problem if it contains more than one statement since it doesn't have a termination symbol.

Example

```
#define macro_name char_sequence
```

The keyword used to define an inline function is "inline" whereas,

the keyword used to define a macro is "#define".

Function Overloading in C++

Function overloading is a feature of object-oriented programming where two or more functions can have the same name but different parameters. When a function name is overloaded with different jobs it is called Function Overloading. In Function Overloading "Function" name should be the same and the arguments should be different. Function overloading can be considered as an example of a polymorphism feature in C++.
The parameters should follow any one or more than one of the following conditions for Function overloading:

- Parameters should have a different type

add(int a, int b)
add(double a, double b)

Below is the implementation of the above discussion:

```
#include <iostream>
using namespace std;
 void add(int a, int b)
{
  cout << "sum = " << (a + b);
}
 void add(double a, double b)
{
   cout << endl << "sum = " << (a + b);
}

// Driver code
int main()
{
   add(10, 2);
   add(5.3, 6.2);

   return 0;
}
```

Output

sum = 12

sum = 11.5

A default argument is **a value provided in a function declaration that is automatically assigned by the compiler if the calling function doesn't provide a value for the argument**. In case any value is passed, the default value is overridden.

Characteristics for defining the default arguments

Following are the rules of declaring default arguments -

- The values passed in the default arguments are not constant. These values can be overwritten if the value is passed to the function. If not, the previously declared value retains.

- During the calling of function, the values are copied from left to right.
- All the values that will be given default value will be on the right.

Example

- void function(int x, int y, int z = 0) Explanation - The above function is valid. Here z is the value that is predefined as a part of the default argument.
- Void function(int x, int z = 0, int y) Explanation - The above function is invalid. Here z is the value defined in between, and it is not accepted.

Code

```
1. #include<iostream>
2. using namespace std;
3. int sum(int x, int y, int z=0, int w=0) // Here there are two values in the default arguments

4. { // Both z and w are initialised to zero
5.    return (x + y + z + w); // return sum of all parameter values
6. }
7. int main()
8. {
9.    cout << sum(10, 15) << endl; // x = 10, y = 15, z = 0, w = 0
10.   cout << sum(10, 15, 25) << endl; // x = 10, y = 15, z = 25, w = 0
11.   cout << sum(10, 15, 25, 30) << endl; // x = 10, y = 15, z = 25, w = 30
12.   return 0;
13. }
```

Output

```
25
50
80
```

Explanation

In the above program, we have called the sum function three times.

- Sum(10,15)
  When this function is called, it reaches the definition of the sum. There it initializes x to 10 y to 15, and the rest values are zero by default as no value is passed. And all the values after sum give 25 as output.
- Sum(10, 15, 25)
  When this function is called, x remains 10, y remains 15, the third parameter z that is passed is initialized to 25 instead of zero. And the last value remains 0. The sum of x, y, z, w, is 50 which is returned as output.
- Sum(10, 15, 25, 30)
  In this function call, there are four parameter values passed into the function with x as 10, y as 15, z is 25, and w as 30. All the values are then summed up to give 80 as the output.

A friend function is **a function that isn't a member of a class but has access to the class's private and protected members**. Friend functions aren't considered class members; they're normal external functions that are given special access privileges.

A friend function is **a special function in C++ which in-spite of not being member function of a class has privilege to access private and protected data of a class**. A friend function is a non member function or ordinary function of a class, which is declared as a friend using the keyword "friend" inside the class

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.

A friend can be a function, function template, or member function, or a class or class template, in which case the entire class and all of its members are friends.

To declare a function as a friend of a class, precede the function prototype in the class definition with keyword **friend** as follows −

```
class Box {
  double width;

  public:
    double length;
    friend void printWidth( Box box );
```

```
    void setWidth( double wid );
};
```

To declare all member functions of class ClassTwo as friends of class ClassOne, place a following declaration in the definition of class ClassOne –

```
friend class ClassTwo;
```

Consider the following program –

```
#include <iostream>

using namespace std;

class Box {
  double width;

  public:
    friend void printWidth( Box box );
    void setWidth( double wid );
};

// Member function definition
void Box::setWidth( double wid ) {
  width = wid;
}

// Note: printWidth() is not a member function of any class.
void printWidth( Box box ) {
  /* Because printWidth() is a friend of Box, it can
  directly access any member of this class */
  cout << "Width of box : " << box.width <<endl;
}

// Main function for the program
int main() {
  Box box;

  // set box width without member function
  box.setWidth(10.0);

  // Use friend function to print the wdith.
  printWidth( box );

  return 0;
}
```

When the above code is compiled and executed, it produces the following result –

Width of box : 10

A virtual function in C++ is **a base class member function that you can redefine in a derived class to achieve polymorphism**. You can declare the function in the base class using the virtual keyword.

A virtual function is nothing but **a member function of a base class that you redefine in a derived class**. The property of redefining a member function declared in the base class to a derived class is also called Function Overriding.

C++ Virtual Functions

In this tutorial, we will learn about C++ virtual function and its use with the help of examples.

A virtual function is a member function in the base class that we expect to redefine in derived classes.

Basically, a virtual function is used in the base class in order to ensure that the function is **overridden**. This especially applies to cases where a pointer of base class points to an object of a derived class.
For example, consider the code below:

```
class Base {
  public:
   void print() {
      // code
   }
};

class Derived : public Base {
  public:
   void print() {
      // code
   }
};
```

Later, if we create a pointer of `Base` type to point to an object of `Derived` class and call the `print()` function, it calls the `print()` function of the `Base` class.
In other words, the member function of `Base` is not overridden.

```
int main() {
  Derived derived1;
  Base* base1 = &derived1;

  // calls function of Base class
  base1->print();

  return 0;
}
```

In order to avoid this, we declare the `print()` function of the `Base` class as virtual by using the `virtual` keyword.

```
class Base {
  public:
   virtual void print() {
     // code
   }
};
```

Virtual functions are an integral part of polymorphism in C++. To learn more, check our tutorial on C++ Polymorphism.