UNIT-I

C++ language is a direct descendant of C programming language with additional features such as type checking, object oriented programming, exception handling etc. You can call it a "better C". It was developed by **Bjarne Stroustrup**.
C++ is a general purpose language, when I say general purpose it simply means that it is designed to be used for developing applications in a wide variety of domains.

Benefits of C++ over C Language

The major difference being OOPS concept, C++ is an object oriented language whereas C language is a procedural language. Apart form this there are many other features of C++ which gives this language an upper hand on C laguage.

Following features of C++ makes it a stronger language than C,

1. There is Stronger Type Checking in C++.
2. All the OOPS features in C++ like Abstraction, Encapsulation, Inheritance etc makes it more worthy and useful for programmers.
3. C++ supports and allows user defined operators (i.e Operator Overloading) and function overloading is also supported in it.
4. Exception Handling is there in C++.
5. The Concept of Virtual functions and also Constructors and Destructors for Objects.
6. Inline Functions in C++ instead of Macros in C language. Inline functions make complete function body act like Macro, safely.
7. Variables can be declared anywhere in the program in C++, but must be declared before they are used.

OOPs Concepts in C++

Object oriented programming is a way of solving complex problems by breaking them into smaller problems using objects. Before Object Oriented Programming (commonly referred as OOP), programs were written in procedural language, they were nothing but a long list of instructions. On the other hand, the OOP is all about creating objects that can interact with each other, this makes it easier to develop programs in OOP as we can understand the relationship between them.

Object Oriented Programming(OOP)

In Object oriented programming we write programs using classes and objects utilising features of OOPs such as **abstraction**, **encapsulation**, **inheritance** and **polymorphism**

Class and Objects

A class is like a blueprint of data member and functions and object is an instance of class. For example, lets say we have a class **Car** which has data members (variables) such as speed, weight, price and functions such as gearChange(), slowDown(), brake() etc. Now lets say I create a object of this class named FordFigo which uses these data members and functions and give them its own values. Similarly we can create as many objects as we want using the blueprint(class).

Abstraction

Abstraction is a process of hiding irrelevant details from user. For example, When you send an sms you just type the message, select the contact and click send, the phone shows you that the message has been sent, what actually happens in background when you click send is hidden from you as it is not relevant to you.

Encapsulation

Encapsulation is a process of combining data and function into a single unit like capsule. This is to avoid the access of private data members from outside the class. To achieve encapsulation, we make all data members of class private and create public functions, using them we can get the values from these data members or set the value to these data members.

Inheritance

Inheritance is a feature using which an object of child class acquires the properties of parent class.

Polymorphism

Function overloading and Operator overloading are examples of polymorphism. Polymorphism is a feature using which an object behaves differently in different situation.
In function overloading we can have more than one function with same name but different numbers, type or sequence of arguments.

C++ Basic Input/Output

C++ Output

In C++, `cout` sends formatted output to standard output devices, such as the screen. We use the `cout` object along with the `<<` operator for displaying output.

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

int main() {
  // prints the string enclosed in double quotes
  cout << "This is C++ Programming";
  return 0;
}
```

How does this program work?

- We first include the `iostream` header file that allows us to display output.
- The `cout` object is defined inside the `std` namespace. To use the `std` namespace, we used the `using namespace std;` statement.
- Every C++ program starts with the `main()` function. The code execution begins from the start of the `main()` function.
- `cout` is an object that prints the string inside quotation marks `" "`. It is followed by the `<<` operator.
- `return 0;` is the "exit status" of the `main()` function. The program ends with this statement, however, this statement is not mandatory.

Note: If we don't include the `using namespace std;` statement, we need to use `std::cout` instead of `cout`.

```
#include <iostream>

int main() {
  // prints the string enclosed in double quotes
  std::cout << "This is C++ Programming";
  return 0;
}
```

Example 2: Numbers and Characters Output

To print the numbers and character variables, we use the same `cout` object but without using quotation marks.

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

int main() {
   int num1 = 70;
   double num2 = 256.783;
   char ch = 'A';

   cout << num1 << endl;    // print integer
   cout << num2 << endl;    // print double
   cout << "character: " << ch << endl;    // print char
   return 0;
}
```

Output

```
70
256.783
character: A
```

Notes:

- The `endl` manipulator is used to insert a new line. That's why each output is displayed in a new line.
- The `<<` operator can be used more than once if we want to print different variables, strings and so on in a single statement. For example:

```
cout << "character: " << ch << endl;
```

C++ Input

In C++, `cin` takes formatted input from standard input devices such as the keyboard. We use the `cin` object along with the `>>` operator for taking input.

Example 3: Integer Input/Output

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

```
int main() {
   int num;
   cout << "Enter an integer: ";
   cin >> num;   // Taking input
   cout << "The number is: " << num;
   return 0;
}
```

Output

```
Enter an integer: 70
The number is: 70
```

In the program, we used

```
cin >> num;
```

to take input from the user. The input is stored in the variable `num`. We use the `>>` operator with `cin` to take input.

Note: If we don't include the `using namespace std;` statement, we need to use `std::cin` instead of `cin`.

C++ Taking Multiple Inputs

```
#include <iostream>
using namespace std;
int main() {
   char a;
   int num;
   cout << "Enter a character and an integer: ";
   cin >> a >> num;
   cout << "Character: " << a << endl;
   cout << "Number: " << num;
   return 0;
}
```

Output

```
Enter a character and an integer: F
23
Character: F
Number: 23
```

The Parts of a C++ Program

The structure of C++ program is divided into four different sections:

(1) Header File Section

(2) Class Declaration section

(3) Member Function definition section

(4) Main function section

(1) Header File Section:

This section contains various header files.

You can include various header files in to your program using this section.

For example:

```
# include <iostream.h >
```

Header file contains declaration and definition of various built in functions as well as object. In order to use this built in functions or object we need to include particular header file in our program.

(2) Class Declaration Section:

This section contains declaration of class.

You can declare class and then declare data members and member functions inside that class.

For example:

```
class Demo
{
int a, b;
public:
void input();
void output();
}
```

You can also inherit one class from another existing class in this section.

(3) Member Function Definition Section:

This section is optional in the structure of C++ program.

Because you can define member functions inside the class or outside the class. If all the member functions are defined inside the class then there is no need of this section.

This section is used only when you want to define member function outside the class.

This section contains definition of the member functions that are declared inside the class.

```
For example:
void Demo:: input ()
{
cout << "Enter Value of A:";
cin >> a;
cout << "Enter Value of B:";
cin >> b;
}
```

(4) Main Function Section:

o In this section you can create an object of the class and then using this object you can call various functions defined inside the class as per your requirement.

```
For example:
Void main ()
{
Demo d1;
d1.input ();
d1.output ();
}
```

We can also compare the structure of C++ program with client server application. In client server application client send request to the server and server sends response to the client.

In above C++ structure the class declaration section and member function definition section both together works as a server and main () function section works as a client. Because in main () function section we create an object of the class and then using that object we make a call to the function declared in the class.

C++ Data Types

In C++, data types are declarations for variables. This determines the type and size of data associated with variables. For example,

```
int age = 13;
```

Here, `age` is a variable of type `int`. Meaning, the variable can only store integers of either 2 or 4 bytes.

C++ Fundamental Data Types

The table below shows the fundamental data types, their meaning, and their sizes (in bytes):

Data Type	Meaning	Size (in Bytes)
`Int`	Integer	2 or 4
`float`	Floating-point	4
`double`	Double Floating-point	8
`char`	Character	1
`wchar_t`	Wide Character	2
`bool`	Boolean	1
`void`	Empty	0

1. C++ int

- The `int` keyword is used to indicate integers.
- Its size is usually 4 bytes. Meaning, it can store values from **-2147483648 to 214748647**.
- For example,

```
int salary = 85000;
```

2. C++ float and double

- `float` and `double` are used to store floating-point numbers (decimals and exponentials).
- The size of `float` is 4 bytes and the size of `double` is 8 bytes. Hence, `double` has two times the precision of `float`. To learn more, visit C++ float and double.
- For example,

```
float area = 64.74;
double volume = 134.64534;
```

As mentioned above, these two data types are also used for exponentials. For example,

```
double distance = 45E12    // 45E12 is equal to 45*10^12
```

3. C++ char

- Keyword `char` is used for characters.
- Its size is 1 byte.
- Characters in C++ are enclosed inside single quotes `''`.

4. C++ bool

- The `bool` data type has one of two possible values: `true` or `false`.
- Booleans are used in conditional statements and loops (which we will learn in later chapters).

```
bool cond = false;
```

5. C++ void

- The `void` keyword indicates an absence of data. It means "nothing" or "no value".

Flow of Control

Control Structures

Control structures are portions of program code that contain statements within them and, depending on the circumstances, execute these statements in a certain way. There are typically two kinds: *conditionals* and *loops*.

2.1 Conditionals

In order for a program to change its behavior depending on the input, there must a way to test that input. Conditionals allow the program to check the values of variables and to execute (or not execute) certain statements. C++ has *if* and *switch-case* conditional structures.

2.1.1 Operators

Conditionals use two kinds of special operators: *relational* and *logical*. These are used to determine whether some condition is true or false.

The relational operators are used to test a relation between two expressions:

Operator	Meaning
>	Greater than
>=	Greater than or equal to
<	Less than
<=	Less than or equal to
==	Equal to
!=	Not equal to

They work the same as the arithmetic operators (e.g., a > b) but return a Boolean value of either true or false, indicating whether the relation tested for holds. (An expression that returns this kind of value is called a Boolean expression.) For example, if the variables x and y have been set to 6 and 2, respectively, then x > y returns true. Similarly, x < 5 returns false.

The logical operators are often used to combine relational expressions into more complicated

Boolean expressions:

Operator	Meaning
&&	and
||	or
!	not

The operators return true or false, according to the rules of logic:

A	**b**	**a && b**
True	true	True
True	false	False
False	true	False
False	false	False

A	**b**	**a || b**
True	true	True
True	false	True
False	true	True
False	false	False

The ! operator is a unary operator, taking only one argument and negating its value:

a	**!a**
true	False
false	True

Examples using logical operators (assume x = 6 and y = 2):

!(x > 2) → false

(x > y) && (y > 0) → true

(x < y) && (y > 0) → false

(x < y) || (y > 0) → true

Of course, Boolean variables can be used directly in these expressions, since they hold true and false values. In fact, any kind of value can be used in a Boolean expression due to a quirk C++ has: false is represented by a value of 0 and anything that is not 0 is true. So, "Hello, world!" is true, 2 is true, and any int variable holding a non-zero value is true. This means !x returns false and x && y returns true!

2.1.2 *if, if-else* and *else if*

The *if* conditional has the form:

if(condition)

```
{
        statement1
        statement2
        …
}
```

The condition is some expression whose value is being tested. If the condition resolves to a value of true, then the statements are executed before the program continues on. Otherwise, the statements are ignored. If there is only one statement, the curly braces may be omitted, giving the form:

```
if(condition)
        statement
```

The *if-else* form is used to decide between two sequences of statements referred to as *blocks*:

```
if(condition)
{
        statementA1
        statementA2
        …
}
else
{
        statementB1
        statementB2
        …
}
```

If the condition is met, the block corresponding to the if is executed. Otherwise, the block corresponding to the else is executed. Because the condition is either satisfied or not, one of the blocks

in an if-else *must* execute. If there is only one statement for any of the blocks, the curly braces for that block may be omitted:

```
if(condition)
        statementA1
else
        statementB1
```

The *else if* is used to decide between two or more blocks based on *multiple* conditions:

```
if(condition1)
{
        statementA1
        statementA2
        …
}
else if(condition2)
{
        statementB1
        statementB2
        …
}
```

If condition1 is met, the block corresponding to the if is executed. If not, then *only if* condition2 is met is the block corresponding to the else if executed. There may be more than one else if, each with its own condition. Once a block whose condition was met is executed, any else ifs after it are ignored. Therefore, in an *if-else-if* structure, either one or no block is executed.

An else may be added to the end of an if-else-if. If none of the previous conditions are met, the else block is executed. In this structure, one of the blocks *must* execute, as in a normal if-else.

Here is an example using these control structures:

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

int main()

{

int x = 6;

int y = 2;

if(x > y)

cout << "x is greater than y\n";

else if(y > x)

cout << "y is greater than x\n";

else

cout << "x and y are equal\n";

return 0;

}
```

The output of this program is x is greater than y. If we replace lines 5 and 6 with

```
int x = 2;

int y = 6;
```

then the output is y is greater than x. If we replace the lines with

```
int x = 2;
int y = 2;
```

then the output is x and y are equal.

2.1.3 s*witch-case*

The *switch-case* is another conditional structure that may or may not execute certain statements. However, the switch-case has peculiar syntax and behavior:

```
switch(expression)
{
        case constant1:
                statementA1

                statementA2
                ...

                break;
        case constant2:

                statementB1
                statementB2

                ...
                break;

        ...
        default:

                statementZ1
                statementZ2

                ...
}
```

The switch evaluates expression and, if expression is equal to constant1, then the statements beneath case constant 1: are executed until a break is encountered. If expression is not equal to constant1, then it is compared to constant2. If these are equal, then the statements beneath case constant 2: are executed until a break is encountered. If not, then the same process repeats for each of the constants, in turn. If none of the constants match, then the statements beneath default: are executed.

Due to the peculiar behavior of switch-cases, curly braces are not necessary for cases where there is more than one statement (but they *are* necessary to enclose the entire switch-case). switch-cases generally have if-else equivalents but can often be a cleaner way of expressing the same behavior.

Here is an example using switch-case:

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

int main() {

int x = 6;

switch(x) {

case 1:

        cout << “x is 1\n”;

        break;

case 2:
case 3:

        cout << "x is 2 or 3";

        break;

default:
                cout << "x is not 1, 2, or 3";

}


return 0;
}
```

This program will print x is not 1, 2, or 3. If we replace line 5 with int x = 2; then the program will print x is 2 or 3.

2.2 Loops

Conditionals execute certain statements if certain conditions are met; loops execute certain statements *while* certain conditions are met. C++ has three kinds of loops: *while, do-while,* and *for*.

2.2.1 w*hile* and *do-while*

The *while* loop has a form similar to the if conditional:

```
while(condition)

{

    statement1

    statement2


    …

}
```

As long as condition holds, the block of statements will be repeatedly executed. If there is only one statement, the curly braces may be omitted. Here is an example:

```
#include <iostream>

using namespace std;

int main() {

int x = 0;

while(x < 10)

x = x + 1;
cout << "x is " << x << "\n";
return 0;
}
```

This program will print x is 10.

The *do-while* loop is a variation that guarantees the block of statements will be executed *at least once:*

```
do

{

        statement1

        statement2

        …

}

while(condition);
```

The block of statements is executed and then, if the condition holds, the program returns to the top of the block. Curly braces are *always* required. Also note the semicolon after the while condition.

2.2.2 *for*

The *for* loop works like the while loop but with some change in syntax:

```
for(initialization; condition; incrementation)

{

        statement1

        statement2

        …

}
```

The for loop is designed to allow a counter variable that is initialized at the beginning of the loop and incremented (or decremented) on each iteration of the loop. Curly braces may be omitted if there is only one statement. Here is an example:

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

int main() {

for(int x = 0; x < 10; x = x + 1)

cout << x << "\n";

return 0;

}
```

This program will print out the values 0 through 9, each on its own line.

If the counter variable is already defined, there is no need to define a new one in the initialization portion of the for loop. Therefore, it is valid to have the following:

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

int main() {

int x = 0;

for(; x < 10; x = x + 1)

cout << x << "\n";

return 0;

}
```

Note that the first semicolon inside the for loop's parentheses is still required.

A for loop can be expressed as a while loop and vice-versa. Recalling that a for loop has the form

```
for(initialization; condition; incrementation)

{
        statement1

        statement2

        …

}
```

we can write an equivalent while loop as

```
while(condition)

{
        statement1

        statement2

        …

        incrementation
}
```

Using our example above,

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

int main() {

for(int x = 0; x < 10; x = x + 1)

cout << x << "\n";

return 0;

}
```

is converted to

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

int main() {

int x = 0;

while(x < 10) {

cout << x << "\n";

x = x + 1;
}

return 0;

}
```

The incrementation step can technically be anywhere inside the statement block, but it is good practice to place it as the last step, particularly if the previous statements use the current value of the counter variable.

2.3 Nested Control Structures

It is possible to place ifs inside of ifs and loops inside of loops by simply placing these structures inside the statement blocks. This allows for more complicated program behavior.

Here is an example using nesting if conditionals:

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

int main() {

int x = 6;

int y = 0;

if(x > y) {

cout << "x is greater than y\n";

if(x == 6)

cout << "x is equal to 6\n";

else

cout << "x is not equalt to 6\n";

} else

cout << "x is not greater than y\n";

return 0;
}
```

This program will print x is greater than y on one line and then x is equal to 6 on the next line.

Here is an example using nested loops:

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

int main() {

for(int x = 0; x < 4; x = x + 1) {

for(int y = 0; y < 4; y = y + 1)

cout << y;

cout << "\n";
}

return 0;

}
```

This program will print four lines of 0123.

C++ Scope resolution operator

The scope resolution operator (::) is used for several reasons. For example: If the global variable name is same as local variable name, the scope resolution operator will be used to call the global variable. It is also used to define a function outside the class and used to access the static variables of class.

Here an example of scope resolution operator in C++ language,

```
#include <iostream>

using namespace std;

char a = 'm';

static int b = 50;

int main() {

  char a = 's';

  cout << "The static variable : "<< ::b;

  cout << "\nThe local variable : " << a;

  cout << "\nThe global variable : " << ::a;

  return 0;

}
```

Here is the output

```
The static variable : 50
The local variable : s
The global variable : m
```

C/C++ Tokens

A token is the smallest element of a program that is meaningful to the compiler. Tokens can be classified as follows:

1. Keywords
2. Identifiers
3. Constants
4. Strings
5. Special Symbols
6. Operators

1. **Keyword:** Keywords are pre-defined or reserved words in a programming language. Each keyword is meant to perform a specific function in a program. Since keywords are referred names for a compiler, they can't be used as variable names because by doing so, we are trying to assign a new meaning to the keyword which is not allowed. You cannot redefine keywords.

However, you can specify text to be substituted for keywords before compilation by using C/C++ preprocessor directives.**C** language supports **32** keywords which are given below:

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

While in **C++** there are **31** additional keywords other than **C** Keywords they are:
asm bool catch class
const_cast delete dynamic_cast explicit
export false friend inline
mutable namespace new operator
private protected public reinterpret_cast
static_cast template this throw
true try typeid typename
using virtual wchar_t

2. **Identifiers:** Identifiers are used as the general terminology for naming of variables, functions and arrays. These are user defined names consisting of arbitrarily long sequence of letters and digits with either a letter or the underscore(_) as a first character. Identifier names must differ in spelling and case from any keywords. You cannot use keywords as identifiers; they are reserved for special use. Once declared, you can use the identifier in later program statements to refer to the associated value. A special kind of identifier, called a statement label, can be used in goto statements.

- They must begin with a letter or underscore(_).
- They must consist of only letters, digits, or underscore. No other special character is allowed.
- It should not be a keyword.
- It must not contain white space.
- It should be up to 31 characters long as only first 31 characters are significant.

3. **Constants:** Constants are also like normal variables. But, only difference is, their values can not be modified by the program once they are defined. Constants refer to fixed values. They are also called as literals. Constants may belong to any of the data type.

Syntax:
const data_type variable_name; (or) **const data_type *variable_name;**

Types of Constants:
1. Integer constants – Example: 0, 1, 1218, 12482
2. Real or Floating point constants – Example: 0.0, 1203.03, 30486.184
3. Octal & Hexadecimal constants – Example: octal: $(013)_8 = (11)_{10}$, Hexadecimal: $(013)_{16} = (19)_{10}$
4. Character constants -Example: 'a', 'A', 'z'
5. String constants -Example: "GeeksforGeeks"

4. **Operators:** Operators are symbols that triggers an action when applied to C variables and other objects. The data items on which operators act upon are called operands. Depending on the number of operands that an operator can act upon, operators can be classified as follows:

Unary Operators: Those operators that require only single operand to act upon are known as unary operators.For Example increment and decrement operators

Binary Operators: Those operators that require two operands to act upon are called binary operators. **Binary operators are classified into :**

1. Arithmetic operators
2. Relational Operators
3. Logical Operators
4. Assignment Operators
5. Conditional Operators
6. Bitwise Operators

Ternary Operators: These operators requires three operands to act upon. For Example Conditional operator(?:).