



1. C++ Tutorial

Tutorial C vs C++ History C++ Features Program C++
cout, cin, endl Variable Data types Keywords
Operators Identifiers 

2. C++ Control Statement

(21-35)

if-else switch For Loop **While Loop**  Do-While Loop
Break Statement Continue Statement Goto
Statement Comments

3. C++ Functions

C++ Functions Call by value & reference C++
Recursion C++ Storage Classes

4. C++ Arrays

Arrays Array to Function Multidimensional Arrays


5. C++ Object Class

C++ OOPs Concepts C++ Object Class C++
Constructor C++ Copy Constructor C++ Destructor C++
this Pointer C++ static C++ Structs C++
Enumeration C++ Friend Function [C++ Math Functions](#)

6. C++ Inheritance

C++ Inheritance  C++ Aggregation

7. C++ Polymorphism

C++ Polymorphism  C++ Overloading [C++
Overriding](#) C++ Virtual Function

8. C++ Strings

C++ Strings

C++ Tutorial

C++ tutorial provides basic and advanced concepts of C++. Our C++ tutorial is designed for beginners and professionals.

C++ is an object-oriented programming language. It is an extension to [C programming](#).

Our C++ tutorial includes all topics of C++ such as first example, control statements, objects and classes, [inheritance](#), [constructor](#), destructor, this, static, polymorphism, abstraction, abstract class, interface, namespace, encapsulation, arrays, strings, exception handling, File IO, etc.

What is C++

C++ is a general purpose, case-sensitive, free-form programming language that supports object-oriented, procedural and generic programming.

C++ is a middle-level language, as it encapsulates both high and low level language features.

Object-Oriented Programming (OOPs)

C++ supports the object-oriented programming, the four major pillar of object-oriented programming ([OOPs](#)) used in C++ are:

1. Inheritance
2. Polymorphism
3. Encapsulation
4. Abstraction

C++ Standard Libraries

Standard C++ programming is divided into three important parts:

- The core library includes the data types, variables and literals, etc.
- The standard library includes the set of functions manipulating strings, files, etc.

- The Standard Template Library (STL) includes the set of methods manipulating a data structure.

Usage of C++

By the help of C++ programming language, we can develop different types of secured and robust applications:

- Window application
- Client-Server application
- Device drivers
- Embedded firmware etc

C++ Program

In this tutorial, all C++ programs are given with C++ compiler so that you can easily change the C++ program code.

File: main.cpp

```
#include <iostream>

using namespace std;

int main() {

    cout << "Hello C++ Programming";

    return 0;

}
```

C vs. C++

What is C?

C is a structural or procedural oriented programming language which is machine-independent and extensively used in various applications.

C is the basic programming language that can be used to develop from the operating systems (like Windows) to complex programs like Oracle database, Git, Python interpreter, and many more. C programming language can be called a god's

programming language as it forms the base for other programming languages. If we know the C language, then we can easily learn other programming languages. C language was developed by the great computer scientist Dennis Ritchie at the Bell Laboratories. It contains some additional features that make it unique from other programming languages.

What is C++?

C++ is a special-purpose programming language developed by **Bjarne Stroustrup** at Bell Labs circa 1980. C++ language is very similar to C language, and it is so compatible with C that it can run 99% of C programs without changing any source of code though C++ is an object-oriented programming language, so it is safer and well-structured programming language than C.

Let's understand the differences between C and C++

The following are the differences between C and C++:

- **Definition**
C is a structural programming language, and it does not support classes and objects, while C++ is an object-oriented programming language that supports the concept of classes and objects.
- **Type of programming language**
C supports the structural programming language where the code is checked line by line, while C++ is an object-oriented programming language that supports the concept of classes and objects.
- **Developer of the language**
Dennis Ritchie developed C language at Bell Laboratories while Bjarne Stroustrup developed the C++ language at Bell Labs circa 1980.
- **Subset**
C++ is a superset of C programming language. C++ can run 99% of C code but C language cannot run C++ code.
- **Type of approach**
C follows the top-down approach, while C++ follows the bottom-up approach. The top-down approach breaks the main modules into tasks; these tasks are broken into sub-tasks, and so on. The bottom-down approach develops the lower level modules first and then the next level modules.
- **Security**
In C, the data can be easily manipulated by the outsiders as it does not

support the encapsulation and information hiding while C++ is a very secure language, i.e., no outsiders can manipulate its data as it supports both encapsulation and data hiding. In C language, functions and data are the free entities, and in C++ language, all the functions and data are encapsulated in the form of objects.

- **Function Overloading**

Function overloading is a feature that allows you to have more than one function with the same name but varies in the parameters. C does not support the function overloading, while C++ supports the function overloading.

- **Function Overriding**

Function overriding is a feature that provides the specific implementation to the function, which is already defined in the base class. C does not support the function overriding, while C++ supports the function overriding.

- **Reference variables**

C does not support the reference variables, while C++ supports the reference variables.

- **Keywords**

C contains 32 keywords, and C++ supports 52 keywords.

- **Namespace feature**

A namespace is a feature that groups the entities like classes, objects, and functions under some specific name. C does not contain the namespace feature, while C++ supports the namespace feature that avoids the name collisions.

- **Exception handling**

C does not provide direct support to the exception handling; it needs to use functions that support exception handling. C++ provides direct support to exception handling by using a try-catch block.

- **Input/Output functions**

In C, scanf and printf functions are used for input and output operations, respectively, while in C++, cin and cout are used for input and output operations, respectively.

- **Memory allocation and de-allocation**

C supports calloc() and malloc() functions for the memory allocation, and free() function for the memory de-allocation. C++ supports a new operator for the memory allocation and delete operator for the memory de-allocation.

- **Inheritance**

Inheritance is a feature that allows the child class to reuse the properties of the parent class. C language does not support the inheritance while C++ supports the inheritance.

- **Headerfile**

C program uses **<stdio.h>** header file while C++ program uses **<iostream.h>** header file.

No.	C	C++
1)	C follows the procedural style programming .	C++ is multi-paradigm. It supports both procedural and object oriented .
2)	Data is less secured in C.	In C++, you can use modifiers for class members to make it inaccessible for outside users.
3)	C follows the top-down approach .	C++ follows the bottom-up approach .
4)	C does not support function overloading.	C++ supports function overloading.
5)	In C, you can't use functions in structure.	In C++, you can use functions in structure.
6)	C does not support reference variables.	C++ supports reference variables.
7)	In C, scanf() and printf() are mainly used for input/output.	C++ mainly uses stream cin and cout to perform input and output operations.
8)	Operator overloading is not possible in C.	Operator overloading is possible in C++.
9)	C programs are divided into procedures and modules	C++ programs are divided into functions and classes .
10)	C does not provide the feature of namespace.	C++ supports the feature of namespace.

11)	Exception handling is not easy in C. It has to perform using other functions.	C++ provides exception handling using Try and Catch block.
12)	C does not support the inheritance.	C++ supports inheritance.

C++ history

History of C++ language is interesting to know. Here we are going to discuss brief history of C++ language.

C++ programming language was developed in 1980 by Bjarne Stroustrup at bell laboratories of AT&T (American Telephone & Telegraph), located in U.S.A.



Bjarne Stroustrup is known as the **founder of C++ language**.

It was develop for adding a feature of **OOP (Object Oriented Programming)** in C without significantly changing the C component.

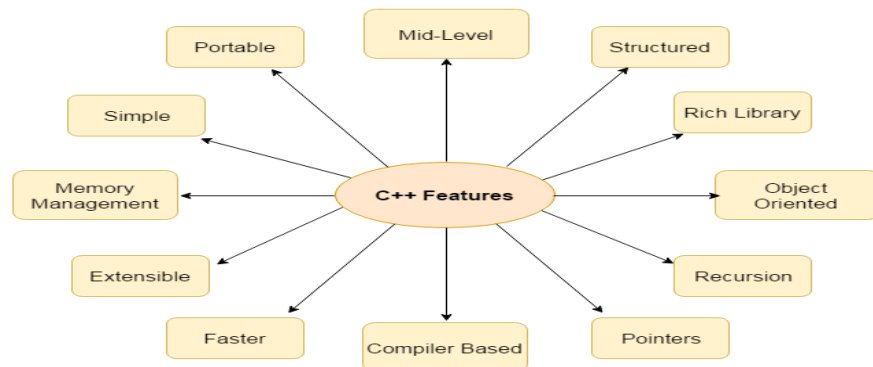
C++ programming is "relative" (called a superset) of C, it means any valid C program is also a valid C++ program.

Let's see the programming languages that were developed before C++ language.

Language	Year	Developed By
Algol	1960	International Group
BCPL	1967	Martin Richard
B	1970	Ken Thompson
Traditional C	1972	Dennis Ritchie
K & R C	1978	Kernighan & Dennis Ritchie
C++	1980	Bjarne Stroustrup

C++ Features

C++ is object oriented programming language. It provides a lot of **features** that are given below.



1. Simple
2. Machine Independent or Portable
3. Mid-level programming language
4. Structured programming language
5. Rich Library
6. Memory Management
7. Fast Speed
8. Pointers
9. Recursion
10. Extensible
11. Object Oriented
12. Compiler based

1) Simple

C++ is a simple language in the sense that it provides structured approach (to break the problem into parts), rich set of library functions, data types etc.

2) Machine Independent or Portable

Unlike assembly language, c programs can be executed in many machines with little bit or no change. But it is not platform-independent.

3) Mid-level programming language

C++ is also used to do low level programming. It is used to develop system applications such as kernel, driver etc. It also supports the feature of high level language. That is why it is known as mid-level language.

4) Structured programming language

C++ is a structured programming language in the sense that we can break the program into parts using functions. So, it is easy to understand and modify.

5) Rich Library

C++ provides a lot of inbuilt functions that makes the development fast.

6) Memory Management

It supports the feature of dynamic memory allocation. In C++ language, we can free the allocated memory at any time by calling the free() function.

7) Speed

The compilation and execution time of C++ language is fast.

8) Pointer

C++ provides the feature of pointers. We can directly interact with the memory by using the pointers. We can use pointers for memory, structures, functions, array etc.

9) Recursion

In C++, we can call the function within the function. It provides code reusability for every function.

10) Extensible

C++ language is extensible because it can easily adopt new features.

11) Object Oriented

C++ is object oriented programming language. OOPs makes development and maintenance easier where as in Procedure-oriented programming language it is not easy to manage if code grows as project size grows.

12) Compiler based

C++ is a compiler based programming language, it means without compilation no C++ program can be executed. First we need to compile our program using compiler and then we can execute our program.

C++ Program

Before starting the abcd of C++ language, you need to learn how to write, compile and run the first C++ program.

To write the first C++ program, open the C++ console and write the following code:

1. `#include <iostream.h>`
2. `#include <conio.h>`
3. `void main() {`
4. `clrscr();`
5. `cout << "Welcome to C++ Programming.";`
6. `getch();`
7. `}`

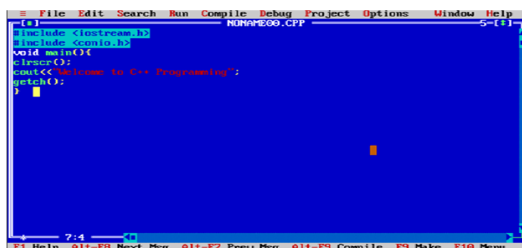
#include <iostream.h> includes the **standard input output** library functions. It provides **cin** and **cout** methods for reading from input and writing to output respectively.

#include <conio.h> includes the **console input output** library functions. The `getch()` function is defined in `conio.h` file.

void main() The **main()** function is the entry point of every program in C++ language. The `void` keyword specifies that it returns no value.

cout << "Welcome to C++ Programming." is used to print the data "Welcome to C++ Programming." on the console.

getch() The `getch()` function **asks for a single character**. Until you press any key, it blocks the screen.



How to compile and run the C++ program

There are 2 ways to compile and run the C++ program, by menu and by shortcut.

By menu

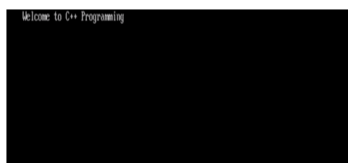
Now **click on the compile menu then compile sub menu** to compile the c++ program.

Then **click on the run menu then run sub menu** to run the c++ program.

By shortcut

Or, press ctrl+f9 keys compile and run the program directly.

You will see the following output on user screen.



You can view the user screen any time by pressing the **alt+f5** keys.

Now **press Esc** to return to the turbo c++ console.

C++ Basic Input/Output

C++ I/O operation is using the stream concept. Stream is the sequence of bytes or flow of data. It makes the performance fast.

If bytes flow from main memory to device like printer, display screen, or a network connection, etc, this is called as **output operation**.

If bytes flow from device like printer, display screen, or a network connection, etc to main memory, this is called as **input operation**.

I/O Library Header Files

Let us see the common header files used in C++ programming are:-

Header File	Function and Description
<iostream>	It is used to define the cout , cin and cerr objects, which correspond to standard output stream, standard input stream and standard error stream, respectively.
<iomanip>	It is used to declare services useful for performing formatted I/O, such as setprecision and setw .
<fstream>	It is used to declare services for user-controlled file processing.

Standard output stream (cout)

The **cout** is a predefined object of **ostream** class. It is connected with the standard output device, which is usually a display screen. The cout is used in conjunction with stream insertion operator (<<) to display the output on a console

Let's see the simple example of standard output stream (cout):

```
1. #include <iostream>
2. using namespace std;
3. int main( )
{
4.     char ary[] = "Welcome to C++ tutorial";
5.     cout << "Value of ary is: " << ary << endl;
6. }
```

Output:

```
Value of ary is: Welcome to C++ tutorial
```

Standard input stream (cin)

The **cin** is a predefined object of **istream** class. It is connected with the standard input device, which is usually a keyboard. The cin is used in conjunction with stream extraction operator (>>) to read the input from a console.

Let's see the simple example of standard input stream (cin):

```
1. #include <iostream>
2. using namespace std;
3. int main() {
4.     int age;
5.     cout << "Enter your age: ";
6.     cin >> age;
7.     cout << "Your age is: " << age << endl;
8. }
```

Output:

```
Enter your age: 22
Your age is: 22
```

Standard end line (endl)

The **endl** is a predefined object of **ostream** class. It is used to insert a new line characters and flushes the stream.

Let's see the simple example of standard end line (endl):

```
1. #include <iostream>
2. using namespace std;
3. int main() {
4.     cout << "C++ Tutorial";
5.     cout << " Javatpoint" << endl;

6.     cout << "End of line" << endl;
7. }
```

Output:

```
C++ Tutorial Javatpoint
End of line
```

C++ Variable

A variable is a name of memory location. It is used to store data. Its value can be changed and it can be reused many times.

It is a way to represent memory location through symbol so that it can be easily identified.

Let's see the syntax to declare a variable:

```
type variable_list;
```

The example of declaring variable is given below:

```
1) int x; 2) float y; 3) char z;
```

Here, x, y, z are variables and int, float, char are data types.

We can also provide values while declaring the variables as given below:

1. **int** x=5,b=10; //declaring 2 variable of integer type
2. **float** f=30.8;
3. **char** c='A';

Rules for defining variables

A variable can have alphabets, digits and underscore.

A variable name can start with alphabet and underscore only. It can't start with digit.

No white space is allowed within variable name.

A variable name must not be any reserved word or keyword e.g. char, float etc.

Valid variable names:

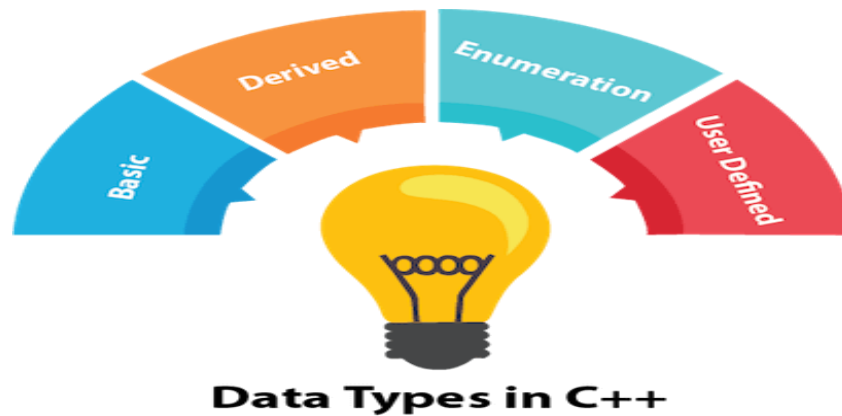
1. **int** a;
2. **int** _ab;
3. **int** a30;

Invalid variable names:

1. **int** 4;
2. **int** x y;
3. **int double**;

C++ Data Types

A data type specifies the type of data that a variable can store such as integer, floating, character etc.



There are 4 types of data types in C++ language.

Types	Data Types
Basic Data Type	int, char, float, double, etc
Derived Data Type	array, pointer, etc
Enumeration Data Type	enum
User Defined Data Type	structure

Basic Data Types

The basic data types are integer-based and floating-point based. C++ language supports both signed and unsigned literals.

The memory size of basic data types may change according to 32 or 64 bit operating system.

Data Types	Memory Size	Range
------------	-------------	-------

char	1 byte	-128 to 127
signed char	1 byte	-128 to 127
unsigned char	1 byte	0 to 127
short	2 byte	-32,768 to 32,767
signed short	2 byte	-32,768 to 32,767
unsigned short	2 byte	0 to 32,767
int	2 byte	-32,768 to 32,767

Let's see the basic data types. Its size is given according to 32 bit OS.

signed int	2 byte	-32,768 to 32,767
unsigned int	2 byte	0 to 32,767
short int	2 byte	-32,768 to 32,767

signed short int	2 byte	-32,768 to 32,767
unsigned short int	2 byte	0 to 32,767
long int	4 byte	
signed long int	4 byte	
unsigned long int	4 byte	
float	4 byte	
double	8 byte	
long double	10 byte	

C++ Keywords

A keyword is a reserved word. You cannot use it as a variable name, constant name etc. A list of 32 Keywords in C++ Language which are also available in C language are given below.

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

asm	dynamic_cast	namespace	reinterpret_cast	bool
explicit	new	static_cast	false	catch
operator	template	friend	private	class
this	inline	public	throw	const_cast
delete	mutable	protected	true	try
typeid	typename	using	virtual	wchar_t

A list of 30 Keywords in C++ Language which are not available in C language are given below.

C++ Operators

An operator is simply a symbol that is used to perform operations. There can be many types of operations like arithmetic, logical, bitwise etc.

There are following types of operators to perform different types of operations in C language.

- Arithmetic Operators
- Relational Operators
- Logical Operators

- Bitwise Operators
- Assignment Operator
- Unary operator
- Ternary or Conditional Operator
- Misc Operator

	Operator	Type
Binary Operator	+, -, *, /, %	Arithmetic Operators
	<, <=, >, >=, ==, !=	Relational Operators
	&&, , !	Logical Operators
	&, , <<, >>, ~, ^	Bitwise Operators
	=, +=, -=, *=, /=, %=	Assignment Operators
Unary Operator	→ ++, --	Unary Operator
Ternary Operator	→ ?:	Ternary or Conditional Operator

Precedence of Operators in C++

The precedence of operator species that which operator will be evaluated first and next. The associativity specifies the operators direction to be evaluated, it may be left to right or right to left.

Let's understand the precedence by the example given below

Category	Operator	Associativity
Postfix	() [] -> . ++ --	Left to right
Unary	+ - ! ~ ++ -- (type)* & sizeof	Right to left
Multiplicative	* / %	Left to right
Additive	+ -	Right to left
Shift	<< >>	Left to right
Relational	< <= > >=	Left to right
Equality	== !=	Right to left

Bitwise AND	&	Left to right
Bitwise XOR	^	Left to right
Bitwise OR		Right to left
Logical AND	&&	Left to right
Logical OR		Left to right
Conditional	?:	Right to left
Assignment	= += -= *= /= %= >>= <<= &= ^= =	Right to left
Comma	,	Left to right

C++ Identifiers

C++ identifiers in a program are used to refer to the name of the variables, functions, arrays, or other user-defined data types created by the programmer. They are the basic requirement of any language. Every language has its own rules for naming the identifiers.

In short, we can say that the C++ identifiers represent the essential elements in a program which are given below:

- **Constants**
- **Variables**
- **Functions**
- **Labels**
- **Defined data types**

Some naming rules are common in both C and C++. They are as follows:

- Only alphabetic characters, digits, and underscores are allowed.
- The identifier name cannot start with a digit, i.e., the first letter should be alphabetical. After the first letter, we can use letters, digits, or underscores.
- In C++, uppercase and lowercase letters are distinct. Therefore, we can say that C++ identifiers are case-sensitive.
- A declared keyword cannot be used as a variable name.

For example, suppose we have two identifiers, named as 'FirstName', and 'Firstname'. Both the identifiers will be different as the letter 'N' in the first case in uppercase while lowercase in second. Therefore, it proves that identifiers are case-sensitive.

Valid Identifiers

The following are the examples of valid identifiers are:

1. Result
2. Test2
3. _sum
4. power

Invalid Identifiers

The following are the examples of invalid identifiers:

1. Sum-1 // containing special character '-'.
2. 2data // the first letter is a digit.
3. **break** // use of a keyword.

Note: Identifiers cannot be used as the keywords. It may not conflict with the keywords, but it is highly recommended that the keywords should not be used as the identifier name. You should always use a consistent way to name the identifiers so that your code will be more readable and maintainable.

The major difference between C and C++ is the limit on the length of the name of the variable. ANSI C considers only the first 32 characters in a name while ANSI C++ imposes no limit on the length of the name.

Constants are the identifiers that refer to the fixed value, which do not change during the execution of a program. Both C and C++ support various kinds of literal constants, and they do have any memory location. For example, 123, 12.34, 037, 0X2, etc. are the literal constants.

Let's look at a simple example to understand the concept of identifiers.

1. **#include** <iostream>
2. **using namespace** std;
3. **int** main()
4. {
5. **int** a;

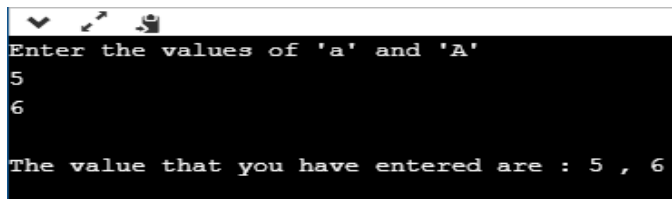
```

6.  int A;
7.  cout<<"Enter the values of 'a' and 'A'";
8.  cin>>a;
9.  cin>>A;
10. cout<<"\nThe values that you have entered are : "<a<<" , "<a<A;
11. return 0;
12.}

```

In the above code, we declare two variables 'a' and 'A'. Both the letters are same but they will behave as different identifiers. As we know that the identifiers are the case-sensitive so both the identifiers will have different memory locations.

Output



```

Enter the values of 'a' and 'A'
5
6
The value that you have entered are : 5 , 6

```

What are the keywords?

Keywords are the reserved words that have a special meaning to the compiler. They are reserved for a special purpose, which cannot be used as the identifiers. For example, 'for', 'break', 'while', 'if', 'else', etc. are the predefined words where predefined words are those words whose meaning is already known by the compiler. Whereas, the identifiers are the names which are defined by the programmer to the program elements such as variables, functions, arrays, objects, classes.

Identifiers	Keywords
Identifiers are the names defined by the programmer to the basic elements of a program.	Keywords are the reserved words whose meaning is known by the compiler.
It is used to identify the name of the variable.	It is used to specify the type of entity.
It can consist of letters, digits, and underscore.	It contains only letters.
It can use both lowercase and uppercase letters.	It uses only lowercase letters.
No special character can be used except the	It cannot contain any special character.

underscore.	
The starting letter of identifiers can be lowercase, uppercase or underscore.	It can be started only with the lowercase letter.
It can be classified as internal and external identifiers.	It cannot be further classified.
Examples are test, result, sum, power, etc.	Examples are 'for', 'if', 'else', 'break', etc.

Differences between Identifiers and Keywords.

The following is the list of differences between identifiers and keywords:

2.CONTROL STATEMENT

C++ if-else

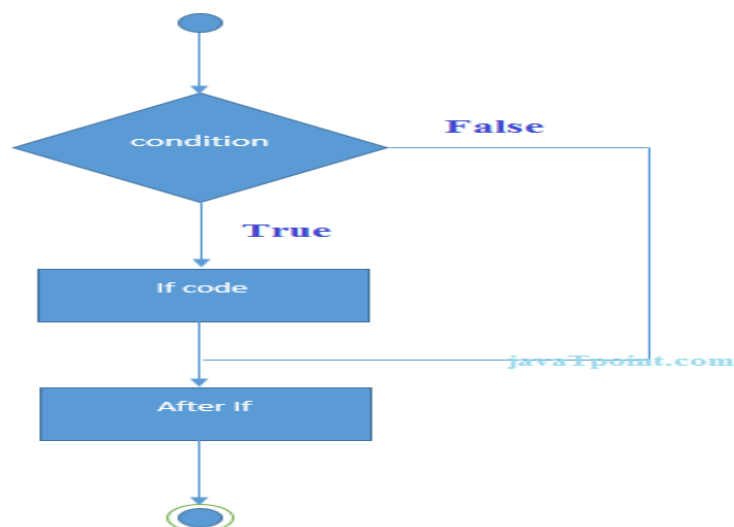
In C++ programming, if statement is used to test the condition. There are various types of if statements in C++.

- if statement
- if-else statement
- nested if statement
- if-else-if ladder

C++ IF Statement

The C++ if statement tests the condition. It is executed if condition is true.

1. **if**(condition){
2. *//code to be executed*
3. }



C++ If Example

1. **#include** <iostream>
2. **using namespace** std;
3. **int** main () {
4. **int** num = 10;

```

5.      if (num % 2 == 0)
6.      {
7.          cout<<"It is even number";
8.      }
9.      return 0;
10. }

```

Output:/p>

```
It is even number
```

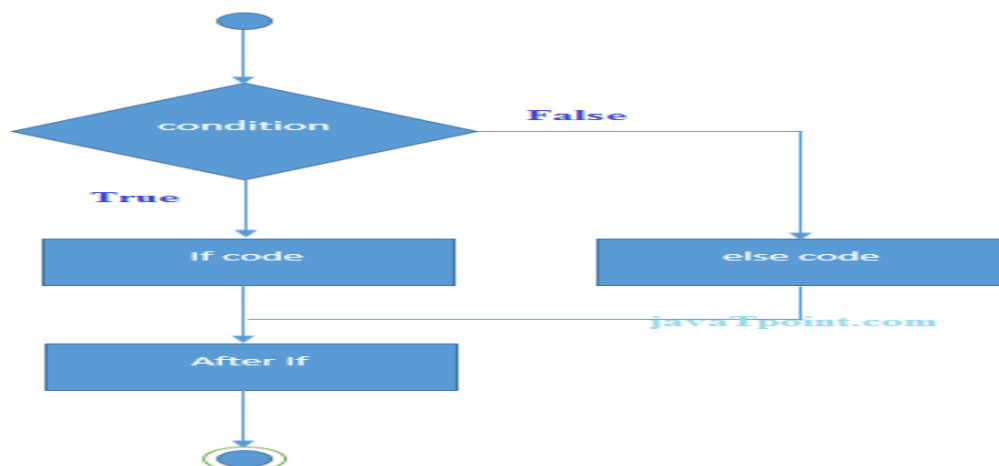
C++ IF-else Statement

The C++ if-else statement also tests the condition. It executes if block if condition is true otherwise else block is executed.

```

1. if(condition){
2.     //code if condition is true
3. }else{
4.     //code if condition is false
5. }

```



C++ If-else Example

```

1. #include <iostream>
2. using namespace std;
3. int main () {
4.     int num = 11;
5.     if (num % 2 == 0)
6.     {

```



```

7.         cout<<"It is even number";
8.     }
9.     else
10.    {
11.        cout<<"It is odd number";
12.    }
13. return 0;

14.}

```

Output:

```
15. It is odd number
```

C++ If-else Example: with input from user

```

1. #include <iostream>
2. using namespace std;
3. int main () {
4.     int num;
5.     cout<<"Enter a Number: ";
6.     cin>>num;
7.     if (num % 2 == 0)
8.     {
9.         cout<<"It is even number"<<endl;
10.    }
11.    else
12.    {
13.        cout<<"It is odd number"<<endl;
14.    }
15. return 0;
16.}

```

Output:

```
Enter a number:11
It is odd number
```

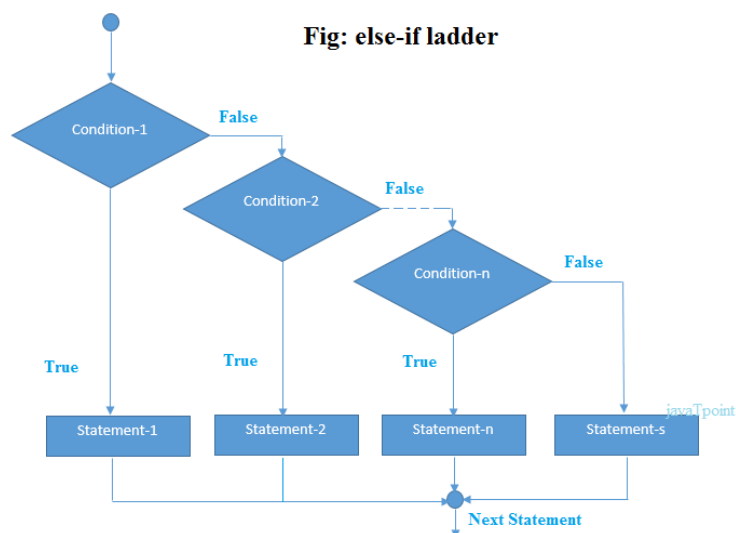
Output:

```
Enter a number:12
It is even number
```

C++ IF-else-if ladder Statement

The C++ if-else-if ladder statement executes one condition from multiple statements.

1. **if**(condition1){
2. //code to be executed if condition1 is true
3. } **else if**(condition2){
4. //code to be executed if condition2 is true
5. }
6. **else if**(condition3){
7. //code to be executed if condition3 is true
8. }
9. ...
10. **else**{
11. //code to be executed if all the conditions are false
12. }



C++ If else-if Example

1. **#include <iostream>**

```

2.  using namespace std;
3.  int main () {
4.      int num;
5.      cout<<"Enter a number to check grade:";
6.      cin>>num;
7.      if (num <0 || num >100)
8.      {

9.          cout<<"wrong number";
10.     }
11.     else if(num >= 0 && num < 50){
12.         cout<<"Fail";
13.     }
14.     else if (num >= 50 && num < 60)
15.     {
16.         cout<<"D Grade";
17.     }
18.     else if (num >= 60 && num < 70)
19.     {
20.         cout<<"C Grade";
21.     }
22.     else if (num >= 70 && num < 80)
23.     {
24.         cout<<"B Grade";
25.     }
26.     else if (num >= 80 && num < 90)
27.     {
28.         cout<<"A Grade";
29.     }
30.     else if (num >= 90 && num <= 100)
31.     {
32.         cout<<"A+ Grade";

33.     }
34. }

```

Output:

```
Enter a number to check grade:66  
C Grade
```

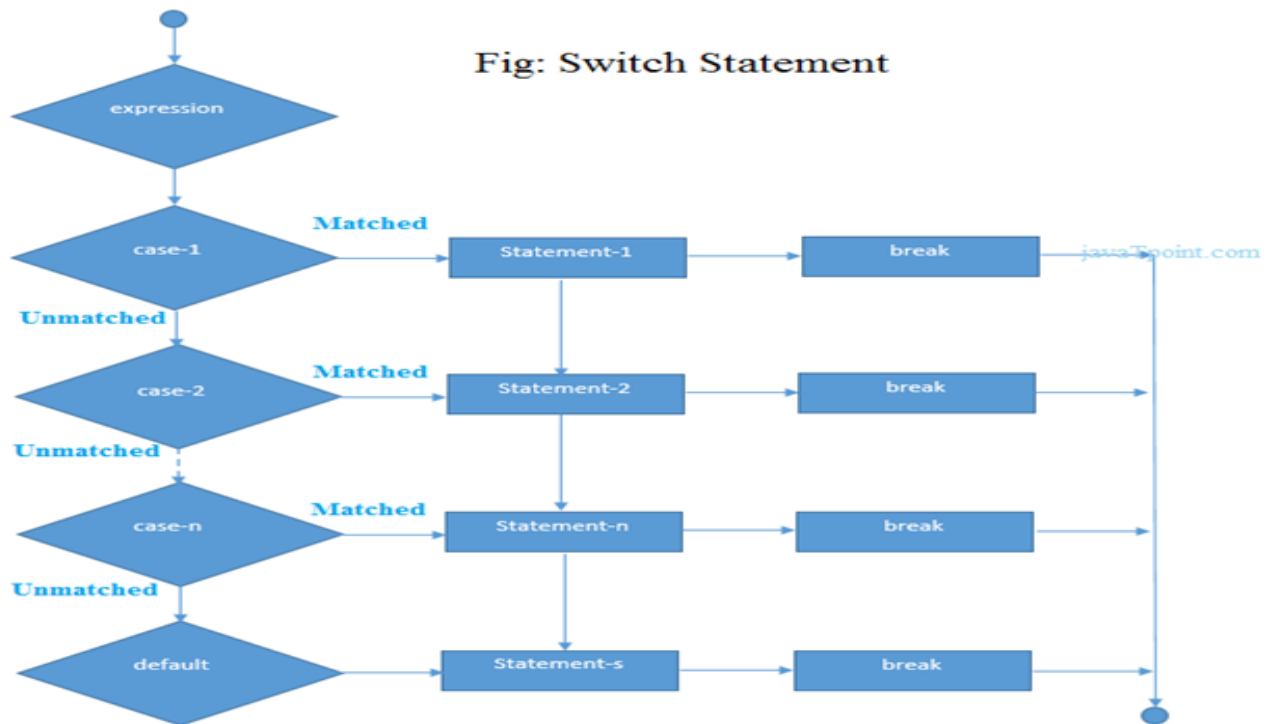
Output:

```
Enter a number to check grade:-2  
wrong number
```

C++ switch

The C++ switch statement executes one statement from multiple conditions. It is like if-else-if ladder statement in C++.

1. **switch**(expression){
2. **case** value1:
3. *//code to be executed;*
4. **break;**
5. **case** value2:
6. *//code to be executed;*
7. **break;**
8.
9. **default:**
10. *//code to be executed if all cases are not matched;*
11. **break;**
12. }



C++ Switch Example

```

1. #include <iostream>
2. using namespace std;
3. int main () {
4.     int num;
5.     cout<<"Enter a number to check grade:";
6.     cin>>num;
7.     switch (num)
8.     {
9.         case 10: cout<<"It is 10"; break;
10.        case 20: cout<<"It is 20"; break;
11.        case 30: cout<<"It is 30"; break;
12.        default: cout<<"Not 10, 20 or 30"; break;
13.    }
14. }
  
```

Output:

```

Enter a number:
10
It is 10
  
```

Output:

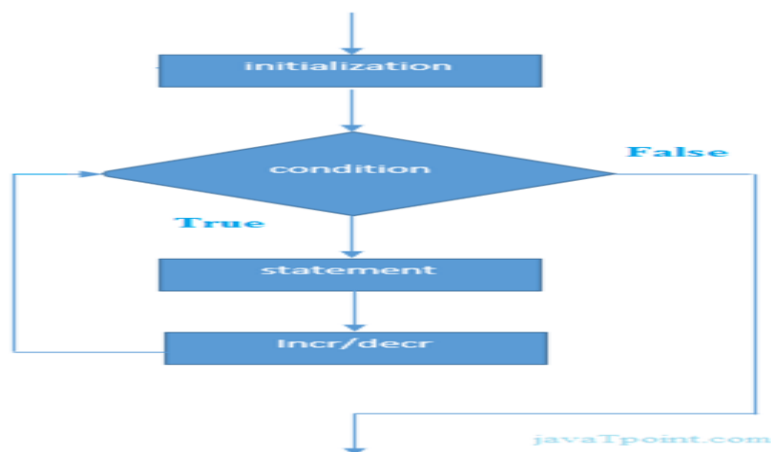
```
Enter a number:  
55  
Not 10, 20 or 30
```

C++ For Loop

The C++ for loop is used to iterate a part of the program several times. If the number of iteration is fixed, it is recommended to use for loop than while or do-while loops.

The C++ for loop is same as C/C#. We can initialize variable, check condition and increment/decrement value.

1. **for**(initialization; condition; incr/decr)
2. {
3. *//code to be executed*
4. }



Flowchart:

C++ For Loop Example

1. `#include <iostream>`
2. `using namespace std;`
3. `int main() {`
4. `for(int i=1;i<=10;i++){`
5. `cout<<i <<"\n";`
6. `}`

7. }

Output:

```
1
2
3
4
5
6
7
8
9
10
```

C++ Nested For Loop

In C++, we can use for loop inside another for loop, it is known as nested for loop. The inner loop is executed fully when outer loop is executed one time. So if outer loop and inner loop are executed 4 times, inner loop will be executed 4 times for each outer loop i.e. total 16 times.

C++ Nested For Loop Example

Let's see a simple example of nested for loop in C++.

```
1. #include <iostream>
2. using namespace std;
3.
4. int main () {
5.     for(int i=1;i<=3;i++){
6.         for(int j=1;j<=3;j++){
7.             cout<<i<<" "<<j<<"\n";
8.         }
9.     }
10. }
```

Output:

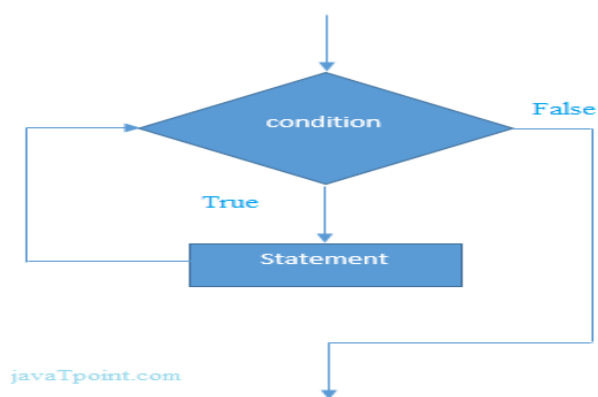
```
1 1
1 2
1 3
```

```
2 1
2 2
2 3
3 1
3 2
3 3
```

C++ While loop

In C++, while loop is used to iterate a part of the program several times. If the number of iteration is not fixed, it is recommended to use while loop than for loop.

1. **while**(condition){
2. *//code to be executed*
3. }



C++ While Loop Example

Let's see a simple example of while loop to print table of 1.

1. **#include** <iostream>
2. **using namespace** std;
3. **int** main() {
4. **int** i=1;
5. **while**(i<=10)
6. {


```

7.      cout<<i <<"\n";
8.      i++;
9.  }
10. }
```

Output:-

```

1
2
3
4
5
6
7
8
9
10
```

C++ Nested While Loop Example

In C++, we can use while loop inside another while loop, it is known as nested while loop. The nested while loop is executed fully when outer loop is executed once.

Let's see a simple example of nested while loop in C++ programming language.

```

1. #include <iostream>
2. using namespace std;
3. int main () {
4.     int i=1;
5.     while(i<=3)
6.     {
7.         int j = 1;
8.         while (j <= 3)
9.         {
10.            cout<<i<<" "<<j<<"\n";
11.            j++;
12.        }
```

```

13.     i++;
14. }
15. }

```

Output:

```

1 1
1 2
1 3
2 1
2 2
2 3
3 1
3 2
3 3

```

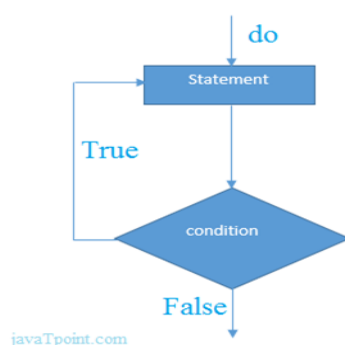
C++ Do-While Loop

The C++ do-while loop is used to iterate a part of the program several times. If the number of iteration is not fixed and you must have to execute the loop at least once, it is recommended to use do-while loop.

The C++ do-while loop is executed at least once because condition is checked after loop body.

1. **do**{
2. *//code to be executed*
3. }**while**(condition);

Flowchart:



C++ do-while Loop Example

Let's see a simple example of C++ do-while loop to print the table of 1.

1. **#include <iostream>**
2. **using namespace std;**

```

3. int main() {
4.     int i = 1;
5.     do{

6.         cout<<i<<"\n";
7.         i++;
8.     } while (i <= 10) ;
9. }

```

Output:

```

1
2
3
4
5
6
7
8
9
10

```

C++ Nested do-while Loop

In C++, if you use do-while loop inside another do-while loop, it is known as nested do-while loop. The nested do-while loop is executed fully for each outer do-while loop.

Let's see a simple example of nested do-while loop in C++.

```

1. #include <iostream>
2. using namespace std;
3. int main() {
4.     int i = 1;
5.     do{
6.         int j = 1;
7.         do{
8.             cout<<i<<"\n";
9.             j++;

```

```

10.     }

11. while (j <= 3);

12.     i++;
13. }
14. while (i <= 3);
15.}

```

Output:

```

1 1
1 2
1 3
2 1
2 2
2 3
3 1
3 2
3 3

```

C++ Break Statement

The C++ break is used to break loop or switch statement. It breaks the current flow of the program at the given condition. In case of inner loop, it breaks only inner loop.

1. jump-statement;
2. **break;**

Flowchart:

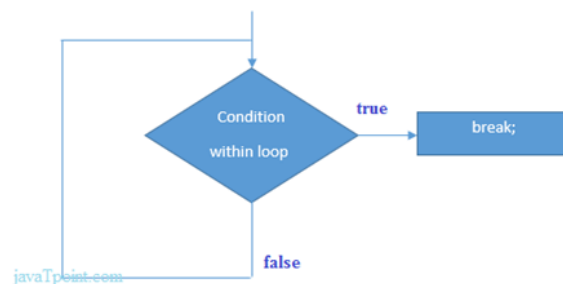


Figure: Flowchart of break statement

C++ Break Statement Example

Let's see a simple example of C++ break statement which is used inside the loop.

```

1. #include <iostream>
2. using namespace std;
3. int main()

4. {
5.     for (int i = 1; i <= 10; i++)
6.     {
7.         if (i == 5)
8.         {
9.             break;
10.        }
11.        cout<<i<<"\n";
12.    }
13.}

```

Output

1 2 3 4

C++ Break Statement with Inner Loop

The C++ break statement breaks inner loop only if you use break statement inside the inner loop.

Let's see the example code:

```

1. #include <iostream>
2. using namespace std;
3. int main()
4. {
5.     for(int i=1;i<=3;i++){
6.         for(int j=1;j<=3;j++){
7.             if(i==2&&j==2){
8.                 break;
9.             }
10.            cout<<i<<" "<<j<<"\n";
11.        }
12.    }

```

13.}

Output:

```
1 1
1 2
1 3
2 1
3 1
3 2
3 3
```

C++ Continue Statement

The C++ continue statement is used to continue loop. It continues the current flow of the program and skips the remaining code at specified condition. In case of inner loop, it continues only inner loop.

1. jump-statement;
2. **continue**;

C++ Continue Statement Example

1. **#include** <iostream>
2. **using namespace** std;
3. **int** main()
4. {
5. **for**(**int** i=1;i<=10;i++){
6. **if**(i==5){
7. **continue**;
8. }
9. cout<<i<<"\n";
10. }
- 11.}

Output:

```
1
2
3
4
6
7
8
9
1
```

C++ Continue Statement with Inner Loop

C++ Continue Statement continues inner loop only if you use continue statement inside the inner loop.

```
1. #include <iostream>
2. using namespace std;
3. int main()
4. {
5.     for(int i=1;i<=3;i++){
6.         for(int j=1;j<=3;j++){
7.             if(i==2&&j==2){
8.                 continue;
9.             }
10.            cout<<i<<" "<j<<"\n";
11.        }
12.    }
13.}
```

Output:

```
1 1
1 2
1 3
2 1
2 3
3 1
3 2
3 3
```

C++ Goto Statement

The C++ goto statement is also known as jump statement. It is used to transfer control to the other part of the program. It unconditionally jumps to the specified label.

It can be used to transfer control from deeply nested loop or switch case label.

C++ Goto Statement Example

Let's see the simple example of goto statement in C++.

```
1. #include <iostream>
```

```

2. using namespace std;
3. int main()
4. {
5.     ineligible:
6.         cout<<"You are not eligible to vote!\n";
7.         cout<<"Enter your age:\n";
8.         int age;
9.         cin>>age;
10.        if (age < 18)
11.        {
12.            goto ineligible;
13.        }
14.        else
15.        {
16.            cout<<"You are eligible to vote!";
17.        }
18.}

```

19. Output:

```

21. You are not eligible to vote!
22. Enter your age:
23. 16
24. You are not eligible to vote!
25. Enter your age:
26. 7
27. You are not eligible to vote!
28. Enter your age:
29. 22
30. You are eligible to vote!

```

C++ Comments

The C++ comments are statements that are not executed by the compiler. The comments in C++ programming can be used to provide explanation of the code, variable, method or class. By the help of comments, you can hide the program code also.

There are two types of comments in C++.

- Single Line comment
- Multi Line comment

C++ Single Line Comment

The single line comment starts with // (double slash). Let's see an example of single line comment in C++.

```
1. #include <iostream>
2. using namespace std;
3. int main()
4. {
5.     int x = 11; // x is a variable
6.     cout<<x<<"\n";
7. }
```

Output:

```
11
```

C++ Multi Line Comment

The C++ multi line comment is used to comment multiple lines of code. It is surrounded by slash and asterisk (/* */). Let's see an example of multi line comment in C++.

```
1. #include <ostream>
2. using namespace std;
3. int main()
4. {
5.     /* declare and
6.     print variable in C++. */
7.     int x = 35;
8.     cout<<x<<"\n";
9. }
```

Output: 35

C++ Functions

The function in C++ language is also known as procedure or subroutine in other programming languages.

To perform any task, we can create function. A function can be called many times. It provides modularity and code reusability.

Advantage of functions in C

There are many advantages of functions.

1) Code Reusability

By creating functions in C++, you can call it many times. So we don't need to write the same code again and again.

2) Code optimization

It makes the code optimized, we don't need to write much code.

Suppose, you have to check 3 numbers (531, 883 and 781) whether it is prime number or not. Without using function, you need to write the prime number logic 3 times. So, there is repetition of code.

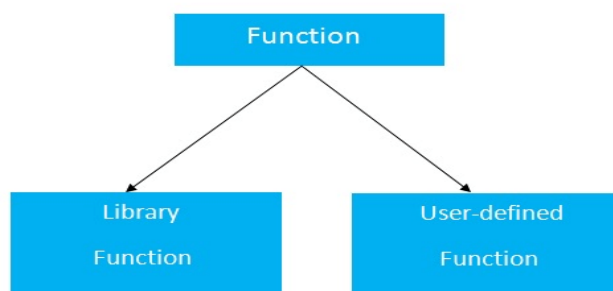
But if you use functions, you need to write the logic only once and you can reuse it several times.

Types of Functions

There are two types of functions in C programming:

1. Library Functions: are the functions which are declared in the C++ header files such as `ceil(x)`, `cos(x)`, `exp(x)`, etc.

2. User-defined functions: are the functions which are created by the C++ programmer, so that he/she can use it many times. It reduces complexity of a big program and optimizes the code.



Declaration of a function

The syntax of creating function in C++ language is given below:

1. return_type function_name(data_type parameter...)
2. {
3. //code to be executed
4. }

C++ Function Example

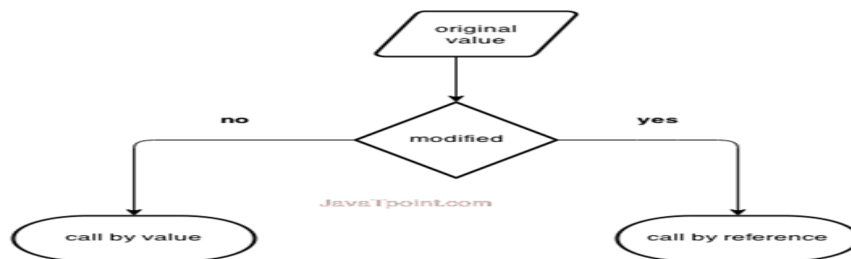
Let's see the simple example of C++ function.

1. `#include <iostream>`
2. `using namespace std;`
3. `void func() {`
4. `static int i=0; //static variable`
5. `int j=0; //local variable`
6. `i++;`
7. `j++;`
8. `cout<<"i=" << i<<" and j=" <<j<<endl;`
9. `}`
10. `int main()`
11. `{`
12. `func();`
13. `func();`
14. `func();`
15. `}`
16. Output:

```
17. i= 1 and j= 1
18. i= 2 and j= 1
19. i= 3 and j= 1
```

2 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.



Let's understand call by value and call by reference in C++ language one by one.

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().

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

```
1. #include <iostream>
2. using namespace std;
3. void change(int data);
4. int main()
5. {
6.     int data = 3;
7.     change(data);
8.     cout << "Value of the data is: " << data << endl;
9.     return 0;
10. }
11. void change(int data)
```

```
12. {  
13. data = 5;  
14. }
```

Output:

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

2 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:

```
1. #include<iostream>  
2. using namespace std;  
3. void swap(int *x, int *y)  
4. {  
5.     int swap;  
6.     swap=*x;  
7.     *x=*y;  
8.     *y=swap;  
9. }  
10. int main()  
11. {  
12.     int x=500, y=100;  
13.     swap(&x, &y); // passing value to function  
14.     cout<<"Value of x is: "<<x<<endl;  
15.     cout<<"Value of y is: "<<y<<endl;  
16.     return 0;
```

17.}

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++ Recursion

When function is called within the same function, it is known as recursion in C++. The function which calls the same function, is known as recursive function.

A function that calls itself, and doesn't perform any task after function call, is known as tail recursion. In tail recursion, we generally call the same function with return statement.

Let's see a simple example of recursion.

1. recursionfunction(){
2. recursionfunction(); //calling self function
3. }

C++ Recursion Example

Let's see an example to print factorial number using recursion in C++ language.

```
1. #include<iostream>
2. using namespace std;
3. int main()
4. {
5.     int factorial(int);
6.     int fact,value;
7.     cout<<"Enter any number: ";
8.     cin>>value;
9.     fact=factorial(value);
10. cout<<"Factorial of a number is: "<<fact<<endl;
11. return 0;
12.}
13. int factorial(int n)
14.{
15. if(n<0)
16. return(-1); /*Wrong value*/
17. if(n==0)
18. return(1); /*Terminating condition*/
19. else
20. {
21. return(n*factorial(n-1));
22.}
23.}
```

Output:

```
Enter any number: 5
Factorial of a number is: 120
```

We can understand the above program of recursive method call by the figure given below:

```

return 5 * factorial(4) = 120
└─ return 4 * factorial(3) = 24
    └─ return 3 * factorial(2) = 6
        └─ return 2 * factorial(1) = 2
            └─ return 1 * factorial(0) = 1

```

javaTpoint.com

$1 * 2 * 3 * 4 * 5 = 120$

Fig: Recursion

C++ Storage Classes

Storage class is used to define the lifetime and visibility of a variable and/or function within a C++ program.

Lifetime refers to the period during which the variable remains active and visibility refers to the module of a program in which the variable is accessible.

There are five types of storage classes, which can be used in a C++ program

1. Automatic
2. Register
3. Static
4. External
5. Mutable

Storage Class	Keyword	Lifetime	Visibility	Initial Value
Automatic	auto	Function Block	Local	Garbage
Register	register	Function Block	Local	Garbage
Mutable	mutable	Class	Local	Garbage
External	extern	Whole Program	Global	Zero
Static	static	Whole Program	Local	Zero

Automatic Storage Class

It is the default storage class for all local variables. The auto keyword is applied to all local variables automatically.

```
1. {  
2. auto int y;  
3. float y = 3.45;  
4. }
```

The above example defines two variables with a same storage class, auto can only be used within functions.

Register Storage Class

The register variable allocates memory in register than RAM. Its size is same of register size. It has a faster access than other variables.

It is recommended to use register variable only for quick access such as in counter.

Note: We can't get the address of register variable.

```
1. register int counter=0;
```

Static Storage Class

The static variable is initialized only once and exists till the end of a program. It retains its value between multiple functions call.

The static variable has the default value 0 which is provided by compiler.

```
1. #include <iostream>  
2. using namespace std;  
3. void func() {  
4.   static int i=0; //static variable  
5.   int j=0; //local variable  
6.   i++;  
7.   j++;  
8.   cout<<"i=" << i<<" and j=" <<j<<endl;  
9. }  
10. int main()  
11. {  
12. func();
```

```
13. func();  
14. func();  
15.}
```

Output:

```
i= 1 and j= 1  
i= 2 and j= 1  
i= 3 and j= 1
```

External Storage Class

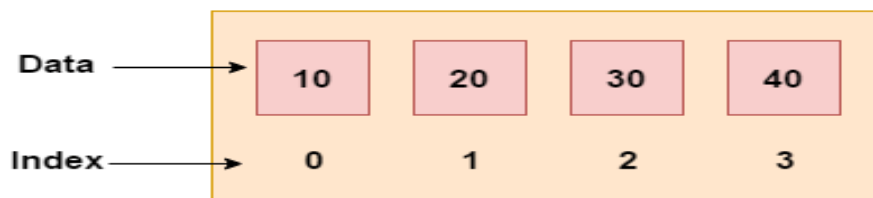
The extern variable is visible to all the programs. It is used if two or more files are sharing same variable or function.

```
extern int counter=0;
```

4. C++ Arrays

Like other programming languages, array in C++ is a group of similar types of elements that have contiguous memory location.

In C++ **std::array** is a container that encapsulates fixed size arrays. In C++, array index starts from 0. We can store only fixed set of elements in C++ array.



Advantages of C++ Array

- Code Optimization (less code)
- Random Access
- Easy to traverse data
- Easy to manipulate data
- Easy to sort data etc.

Disadvantages of C++ Array

- Fixed size

C++ Array Types

There are 2 types of arrays in C++ programming:

1. Single Dimensional Array
2. Multidimensional Array

C++ Single Dimensional Array

Let's see a simple example of C++ array, where we are going to create, initialize and traverse array.

1. `#include <iostream>`
2. `using namespace std;`

```

3. int main()
4. {
5.     int arr[5]={10, 0, 20, 0, 30}; //creating and initializing array
6.         //traversing array
7.     for (int i = 0; i < 5; i++)
8.     {
9.         cout<<arr[i]<<"\n";
10.    }
11.}

```

Output:/p>

```

10
0
20
0
30

```

C++ Array Example: Traversal using foreach loop

We can also traverse the array elements using foreach loop. It returns array element one by one.

```

1. #include <iostream>
2. using namespace std;
3. int main()
4. {
5.     int arr[5]={10, 0, 20, 0, 30}; //creating and initializing array
6.         //traversing array
7.     for (int i: arr)
8.     {
9.         cout<<i<<"\n";
10.    }
11.}

```

Output:

```

10
20
30
40
50

```

C++ Passing Array to Function

In C++, to reuse the array logic, we can create function. To pass array to function in C++, we need to provide only array name.

1. `functionname(arrayname); //passing array to function`

C++ Passing Array to Function Example: print array elements

Let's see an example of C++ function which prints the array elements.

```
1. #include <iostream>
2. using namespace std;
3. void printMin(int arr[5]);
4. int main()
5. {
6.     int arr1[5] = { 30, 10, 20, 40, 50 };
7.     int arr2[5] = { 5, 15, 25, 35, 45 };
8.     printMin(arr1); //passing array to function
9.     printMin(arr2);
10.}
11. void printMin(int arr[5])
12. {
13.     int min = arr[0];
14.     for (int i = 0; i < 5; i++)
15.     {
16.         if (min > arr[i])
17.         {
18.             min = arr[i];
19.         }
20.     }
21.     cout << "Minimum element is: " << min << "\n";
22.}
```

Output:

```
Minimum element is: 10
Minimum element is: 5
```

C++ Passing Array to Function Example: Print maximum number

Let's see an example of C++ array which prints maximum number in an array using function.

```
1. #include <iostream>
2. using namespace std;
3. void printMax(int arr[5]);
4. int main()
5. {
6.     int arr1[5] = { 25, 10, 54, 15, 40 };
7.     int arr2[5] = { 12, 23, 44, 67, 54 };
8.     printMax(arr1); //Passing array to function
9.     printMax(arr2);
10.}
11. void printMax(int arr[5])
12. {
13.     int max = arr[0];
14.     for (int i = 0; i < 5; i++)
15.     {
16.         if (max < arr[i])
17.         {
18.             max = arr[i];
19.         }
20.     }
21.     cout<< "Maximum element is: "<< max <<"\n";
22.}
```

Output:

```
Maximum element is: 54
Maximum element is: 67
```

C++ Multidimensional Arrays

The multidimensional array is also known as rectangular arrays in C++. It can be two dimensional or three dimensional. The data is stored in tabular form (row * column) which is also known as matrix.

C++ Multidimensional Array Example

Let's see a simple example of multidimensional array in C++ which declares, initializes and traverse two dimensional arrays.

```
1. #include <iostream>
2. using namespace std;
3. int main()
4. {
5.     int test[3][3]; //declaration of 2D array
6.     test[0][0]=5; //initialization
7.     test[0][1]=10;
8.     test[1][1]=15;
9.     test[1][2]=20;
10.    test[2][0]=30;
11.    test[2][2]=10;
12.    //traversal
13.    for(int i = 0; i < 3; ++i)
14.    {
15.        for(int j = 0; j < 3; ++j)
16.        {
17.            cout<< test[i][j]<<" ";
18.        }
19.        cout<<"\n"; //new line at each row
20.    }
21.    return 0;
22.}
```

Output:

```
5 10 0
0 15 20
30 0 10
```

C++ Multidimensional Array Example: Declaration and initialization at same time

Let's see a simple example of multidimensional array which initializes array at the time of declaration.

```
1. #include <iostream>
2. using namespace std;
3. int main()
4. {
5.     int test[3][3] =
6.     {
7.         {2, 5, 5},
8.         {4, 0, 3},
9.         {9, 1, 8} }; //declaration and initialization
10.    //traversal
11.    for(int i = 0; i < 3; ++i)
12.    {
13.        for(int j = 0; j < 3; ++j)
14.        {
15.            cout<< test[i][j]<<" ";
16.        }
17.        cout<<"\n"; //new line at each row
18.    }
19.    return 0;
20.}
```

Output:"

```
2 5 5
4 0 3
9 1 8
```


5.CHAPTER

C++ OOPs Concepts

The major purpose of C++ programming is to introduce the concept of object orientation to the C programming language.

Object Oriented Programming is a paradigm that provides many concepts such as **inheritance, data binding, polymorphism etc.**

The programming paradigm where everything is represented as an object is known as truly object-oriented programming language. **Smalltalk** is considered as the first truly object-oriented programming language.

OOPs (Object Oriented Programming System)

Object means a real word entity such as pen, chair, table etc. **Object-Oriented Programming** is a methodology or paradigm to design a program using classes and objects. It simplifies the software development and maintenance by providing some concepts:

- Object
- Class
- Inheritance
- Polymorphism
- Abstraction
- Encapsulation

Object

Any entity that has state and behavior is known as an object. For example: chair, pen, table, keyboard, bike etc. It can be physical and logical.

Class

Collection of objects is called class. It is a logical entity

Inheritance

When one object acquires all the properties and behaviours of parent object i.e. known as inheritance. It provides code reusability. It is used to achieve runtime polymorphism.

Polymorphism

When **one task is performed by different ways** i.e. known as polymorphism. For example: to convince the customer differently, to draw something e.g. shape or rectangle etc.

In C++, we use Function overloading and Function overriding to achieve polymorphism.

Abstraction

Hiding internal details and showing functionality is known as abstraction. For example: phone call, we don't know the internal processing.

In C++, we use abstract class and interface to achieve abstraction.

Encapsulation

Binding (or wrapping) code and data together into a single unit is known as encapsulation. For example: capsule, it is wrapped with different medicines.

Advantage of OOPs over Procedure-oriented programming language

1. OOPs makes development and maintenance easier where as in Procedure-oriented programming language it is not easy to manage if code grows as project size grows.
2. OOPs provide data hiding whereas in Procedure-oriented programming language a global data can be accessed from anywhere.
3. OOPs provide ability to simulate real-world event much more effectively. We can provide the solution of real word problem if we are using the Object-Oriented Programming language.

C++ Object and Class

Since C++ is an object-oriented language, program is designed using objects and classes in C++

C++ Object

In C++, Object is a real world entity, for example, chair, car, pen, mobile, laptop etc.

In other words, object is an entity that has state and behavior. Here, state means data and behavior means functionality.

Object is a runtime entity, it is created at runtime.

Object is an instance of a class. All the members of the class can be accessed through object.

Let's see an example to create object of student class using s1 as the reference variable.

1. Student s1; //creating an object of Student

In this example, Student is the type and s1 is the reference variable that refers to the instance of Student class.

C++ Class

In C++, class is a group of similar objects. It is a template from which objects are created. It can have fields, methods, constructors etc.

Let's see an example of C++ class that has three fields only.

1. **class** Student
2. {
3. **public:**
4. **int** id; //field or data member
5. **float** salary; //field or data member
6. String name; //field or data member
7. }

C++ Object and Class Example

Let's see an example of class that has two fields: id and name. It creates instance of the class, initializes the object and prints the object value.

1. **#include** <iostream>
2. **using namespace** std;
3. **class** Student {
4. **public:**
5. **int** id; //data member (also instance variable)
6. string name; //data member(also instance variable)
7. };
8. **int** main() {

```

9.   Student s1; //creating an object of Student
10.  s1.id = 201;
11.  s1.name = "Sonoo Jaiswal";
12.  cout<<s1.id<<endl;
13.  cout<<s1.name<<endl;
14.  return 0;
15.}

```

Output:

```

16.  201
17.  Sonoo Jaiswal

```

C++ Class Example: Initialize and Display data through method

Let's see another example of C++ class where we are initializing and displaying object through method.

```

1.  #include <iostream>
2.  using namespace std;
3.  class Student {
4.      public:
5.          int id;//data member (also instance variable)
6.          string name;//data member(also instance variable)
7.          void insert(int i, string n)
8.          {
9.              id = i;
10.             name = n;
11.         }
12.         void display()
13.         {
14.             cout<<id<<" "<<name<<endl;
15.         }
16. };
17. int main(void) {
18.     Student s1; //creating an object of Student
19.     Student s2; //creating an object of Student
20.     s1.insert(201, "Sonoo");
21.     s2.insert(202, "Nakul");
22.     s1.display();

```

```
23. s2.display();
24. return 0;
25. }
```

Output:

```
26. 201 Sonoo
27. 202 Nakul
```

C++ Class Example: Store and Display Employee Information

Let's see another example of C++ class where we are storing and displaying employee information using method.

```
1. #include <iostream>
2. using namespace std;
3. class Employee {
4.     public:
5.         int id;//data member (also instance variable)
6.         string name;//data member(also instance variable)
7.         float salary;
8.         void insert(int i, string n, float s)
9.         {
10.            id = i;
11.            name = n;
12.            salary = s;
13.        }
14.        void display()
15.        {
16.            cout<<id<<" "<<name<<" "<<salary<<endl;
17.        }
18. };
19. int main(void) {
20.     Employee e1; //creating an object of Employee
21.     Employee e2; //creating an object of Employee
22.     e1.insert(201, "Sonoo",990000);
23.     e2.insert(202, "Nakul", 29000);
24.     e1.display();
25.     e2.display();
```

```
26. return 0;
27. }
```

Output:

```
28. 201 Sonoo 990000
29. 202 Nakul 29000
```

C++ Constructor

In C++, constructor is a special method which is invoked automatically at the time of object creation. It is used to initialize the data members of new object generally. The constructor in C++ has the same name as class or structure.

There can be two types of constructors in C++.

- Default constructor
- Parameterized constructor

C++ Default Constructor

A constructor which has no argument is known as default constructor. It is invoked at the time of creating object.

Let's see the simple example of C++ default Constructor.

```
1. #include <iostream>
2. using namespace std;
3. class Employee
4. {
5.     public:
6.         Employee()
7.         {
8.             cout<<"Default Constructor Invoked"<<endl;
9.         }
10. };
11. int main(void)
12. {
13.     Employee e1; //creating an object of Employee
14.     Employee e2;
15.     return 0;
```

16.}

Output:

```
Default Constructor Invoked  
Default Constructor Invoked
```

C++ Parameterized Constructor

A constructor which has parameters is called parameterized constructor. It is used to provide different values to distinct objects.

Let's see the simple example of C++ Parameterized Constructor.

```
#include <iostream>  
using namespace std;  
class Employee {  
    public:  
        int id;//data member (also instance variable)  
        string name;//data member(also instance variable)  
        float salary;  
        Employee(int i, string n, float s)  
        {  
            id = i;  
            name = n;  
            salary = s;  
        }  
        void display()  
        {  
            cout<<id<<" "<<name<<" "<<salary<<endl;  
        }  
};  
int main(void) {  
    Employee e1 =Employee(101, "Sonoo", 890000); //creating an object of  
Employee  
    Employee e2=Employee(102, "Nakul", 59000);  
    e1.display();  
    e2.display();  
    return 0;  
}
```

Output:

```
Sonoo 890000  
Nakul 59000
```

C++ Destructor

A destructor works opposite to constructor; it destructs the objects of classes. It can be defined only once in a class. Like constructors, it is invoked automatically.

A destructor is defined like constructor. It must have same name as class. But it is prefixed with a tilde sign (~).

Note: C++ destructor cannot have parameters. Moreover, modifiers can't be applied on destructors.

C++ Constructor and Destructor Example

Let's see an example of constructor and destructor in C++ which is called automatically.

```
1. #include <iostream>
2. using namespace std;
3. class Employee
4. {
5.     public:
6.         Employee()
7.         {
8.             cout<<"Constructor Invoked"<<endl;
9.         }
10.        ~Employee()
11.        {
12.            cout<<"Destructor Invoked"<<endl;
13.        }
14. };
15. int main(void)
16. {
17.     Employee e1; //creating an object of Employee
18.     Employee e2; //creating an object of Employee
19.     return 0;
20. }
```

Output:

```
Constructor Invoked
Constructor Invoked
Destructor Invoked
Destructor Invoked
```

C++ this Pointer

In C++ programming, **this** is a keyword that refers to the current instance of the class. There can be 3 main usage of this keyword in C++.

- It can be used **to pass current object as a parameter to another method.**
- It can be used **to refer current class instance variable.**
- It can be used **to declare indexers.**

C++ this Pointer Example

Let's see the example of this keyword in C++ that refers to the fields of current class.

```

1. #include <iostream>
2. using namespace std;
3. class Employee {
4.     public:
5.         int id; //data member (also instance variable)
6.         string name; //data member(also instance variable)
7.         float salary;
8.         Employee(int id, string name, float salary)
9.         {
10.            this->id = id;
11.            this->name = name;
12.            this->salary = salary;
13.        }
14.        void display()
15.        {
16.            cout<<id<<" "<<name<<" "<<salary<<endl;
17.        }
18. };
19. int main(void) {
20.     Employee e1 =Employee(101, "Sonoo", 890000); //creating an object of Employee
21.     Employee e2=Employee(102, "Nakul", 59000); //creating an object of Employee
22.     e1.display();
23.     e2.display();
24.     return 0;
25. }
```

Output:

```
Sonoo  890000
```

C++ static

In C++, static is a keyword or modifier that belongs to the type not instance. So instance is not required to access the static members. In C++, static can be field, method, constructor, class, properties, operator and event.

Advantage of C++ static keyword

Memory efficient: Now we don't need to create instance for accessing the static members, so it saves memory. Moreover, it belongs to the type, so it will not get memory each time when instance is created.

C++ Structs

In C++, classes and structs are blueprints that are used to create the instance of a class. Structs are used for lightweight objects such as Rectangle, color, Point, etc.

Unlike class, structs in C++ are value type than reference type. It is useful if you have data that is not intended to be modified after creation of struct.

C++ Structure is a collection of different data types. It is similar to the class that holds different types of data.

Structure v/s Class

Structure	Class
If access specifier is not declared explicitly, then by default access specifier will be public.	If access specifier is not declared explicitly, then by default access specifier will be private.
Syntax of Structure: <pre>struct structure_name { // body of the structure. }</pre>	Syntax of Class: <pre>class class_name { // body of the class. }</pre>
The instance of the structure is known as "Structure variable".	The instance of the class is known as "Object of the class".

The Syntax Of Structure

```
1. struct structure_name
2. {
3.     // member declarations.
4. }

#include <iostream>
using namespace std;
struct Rectangle
{
    int width, height;

};
int main(void) {
    struct Rectangle rec;
    rec.width=8;
    rec.height=5;
    cout<<"Area of Rectangle is: "<<(rec.width * rec.height)<<endl;
    return 0;
}
```

Output:

```
Area of Rectangle is: 40
```

C++ Friend function

If a function is defined as a friend function in C++, then the protected and private data of a class can be accessed using the function.

By using the keyword friend compiler knows the given function is a friend function.

For accessing the data, the declaration of a friend function should be done inside the body of a class starting with the keyword friend.

Declaration of friend function in C++

```
1. class class_name
2. {
3.     friend data_type function_name(argument/s);    // syntax of friend function.
4. };
```

In the above declaration, the friend function is preceded by the keyword `friend`. The function can be defined anywhere in the program like a normal C++ function. The function definition does not use either the keyword **friend** or **scope resolution operator**.

Characteristics of a Friend function:

- The function is not in the scope of the class to which it has been declared as a friend.
- It cannot be called using the object as it is not in the scope of that class.
- It can be invoked like a normal function without using the object.
- It cannot access the member names directly and has to use an object name and dot membership operator with the member name.
- It can be declared either in the private or the public part.

C++ friend function Example

Let's see the simple example of C++ friend function used to print the length of a box.

```
1. #include <iostream>
2. using namespace std;
3. class Box
4. {
5.     private:
6.         int length;
7.     public:
8.         Box(): length(0) { }
9.         friend int printLength(Box); //friend function
10. };
11. int printLength(Box b)
12. {
13.     b.length += 10;
14.     return b.length;
15. }
16. int main()
17. {
18.     Box b;
19.     cout<<"Length of box: "<< printLength(b)<<endl;
20.     return 0;
```

21. }

Output:

```
Length of box: 10
```

C++ Friend class

A friend class can access both private and protected members of the class in which it has been declared as friend.

Let's see a simple example of a friend class.

```
1. #include <iostream>
2.
3. using namespace std;
4.
5. class A
6. {
7.     int x =5;
8.     friend class B;        // friend class.
9. };
10. class B
11. {
12. public:
13.     void display(A &a)
14.     {
15.         cout<<"value of x is : "<<a.x;
16.     }
17. };
18. int main()
19. {
20.     A a;
21.     B b;
22.     b.display(a);
23.     return 0;
24. }
```

Output:

```
value of x is : 5
```

In the above example, class B is declared as a friend inside the class A. Therefore, B is a friend of class A. Class B can access the private members of class A.

C++ Math Functions

C++ offers some basic math functions and the required header file to use these functions is `<math.h>`

Trigonometric functions

Method	Description
<code>cos(x)</code>	It computes the cosine of x.
<code>sin(x)</code>	It computes the sine of x.
<code>tan(x)</code>	It computes the tangent of x.
<code>acos(x)</code>	It finds the inverse cosine of x.
<code>asin(x)</code>	It finds the inverse sine of x.
<code>atan(x)</code>	It finds the inverse tangent of x.
<code>atan2(x,y)</code>	It finds the inverse tangent of a coordinate x and y.

CHAPTER 6

C++ Inheritance

In C++, inheritance is a process in which one object acquires all the properties and behaviors of its parent object automatically. In such way, you can reuse, extend or modify the attributes and behaviors which are defined in other class.

In C++, the class which inherits the members of another class is called derived class and the class whose members are inherited is called base class. The derived class is the specialized class for the base class.

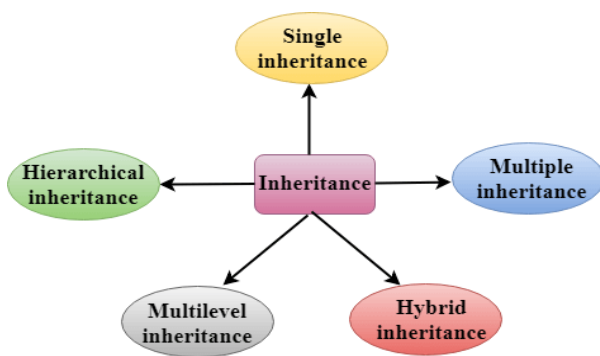
Advantage of C++ Inheritance

Code reusability: Now you can reuse the members of your parent class. So, there is no need to define the member again. So less code is required in the class.

Types Of Inheritance

C++ supports five types of inheritance:

- Single inheritance
- Multiple inheritance
- Hierarchical inheritance
- Multilevel inheritance
- Hybrid inheritance



Derived Classes

A Derived class is defined as the class derived from the base class.

The Syntax of Derived class:

1. **class** derived_class_name :: visibility-mode base_class_name
2. {
3. // body of the derived class.
4. }

Where,

derived_class_name: It is the name of the derived class.

visibility mode: The visibility mode specifies whether the features of the base class are publicly inherited or privately inherited. It can be public or private.

base_class_name: It is the name of the base class.

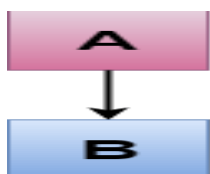
- When the base class is privately inherited by the derived class, public members of the base class become the private members of the derived class. Therefore, the public members of the base class are not accessible by the objects of the derived class only by the member functions of the derived class.
- When the base class is publicly inherited by the derived class, public members of the base class also become the public members of the derived class. Therefore, the public members of the base class are accessible by the objects of the derived class as well as by the member functions of the base class.

Note:

- In C++, the default mode of visibility is private.
- The private members of the base class are never inherited.

C++ Single Inheritance

Single inheritance is defined as the inheritance in which a derived class is inherited from the only one base class.



Where 'A' is the base class, and 'B' is the derived class.

C++ Single Level Inheritance Example: Inheriting Fields

When one class inherits another class, it is known as single level inheritance. Let's see the example of single level inheritance which inherits the fields only.

```
1. #include <iostream>
2. using namespace std;
3. class Account {
4.     public:
5.         float salary = 60000;
6. };
7. class Programmer: public Account {
8.     public:
9.         float bonus = 5000;
10. };
11. int main(void) {
12.     Programmer p1;
13.     cout<<"Salary: "<<p1.salary<<endl;
14.     cout<<"Bonus: "<<p1.bonus<<endl;
15.     return 0;
16. }
```

Output:

```
Salary: 60000
Bonus: 5000
```

In the above example, Employee is the **base** class and Programmer is the **derived** class.

C++ Single Level Inheritance Example: Inheriting Methods

Let's see another example of inheritance in C++ which inherits methods only.

```
1. #include <iostream>
2. using namespace std;
3. class Animal {
4.     public:
5.     void eat() {
6.         cout<<"Eating..."<<endl;
7.     }
8. };
9. class Dog: public Animal
10. {
11.     public:
12.     void bark(){
13.         cout<<"Barking...";
14.     }
15. };
16. int main(void) {
17.     Dog d1;
18.     d1.eat();
19.     d1.bark();
20.     return 0;
21. }
```

Output:

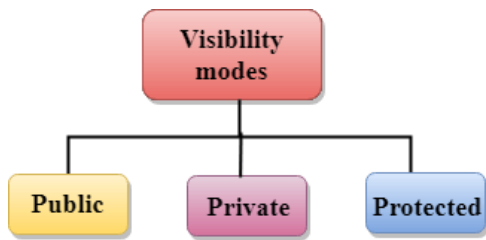
```
Eating...
Barking...
```

How to make a Private Member Inheritable

The private member is not inheritable. If we modify the visibility mode by making it public, but this takes away the advantage of data hiding.

C++ introduces a third visibility modifier, i.e., **protected**. The member which is declared as protected will be accessible to all the member functions within the class as well as the class immediately derived from it.

Visibility modes can be classified into three categories:



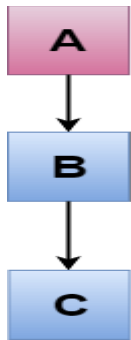
- **Public:** When the member is declared as public, it is accessible to all the functions of the program.
- **Private:** When the member is declared as private, it is accessible within the class only.
- **Protected:** When the member is declared as protected, it is accessible within its own class as well as the class immediately derived from it.

Visibility of Inherited Members

Base class visibility	Derived class visibility		
	Public	Private	Protected
Private	Not Inherited	Not Inherited	Not Inherited
Protected	Protected	Private	Protected
Public	Public	Private	Protected

C++ Multilevel Inheritance

Multilevel inheritance is a process of deriving a class from another derived class.



C++ Multi Level Inheritance Example

When one class inherits another class which is further inherited by another class, it is known as multi level inheritance in C++. Inheritance is transitive so the last derived class acquires all the members of all its base classes.

Let's see the example of multi level inheritance in C++.

```
1. #include <iostream>
2. using namespace std;
3. class Animal {
4.     public:
5.     void eat() {
6.         cout<<"Eating..."<<endl;
7.     }
8. };
9. class Dog: public Animal
10. {
11.     public:
12.     void bark(){
13.         cout<<"Barking..."<<endl;
14.     }
15. };
16. class BabyDog: public Dog
17. {
18.     public:
19.     void weep() {
20.         cout<<"Weeping..."
```

```

21. }
22. };
23. int main(void) {
24.     BabyDog d1;
25.     d1.eat();
26.     d1.bark();
27.     d1.weep();
28.     return 0;
29. }

```

Output:

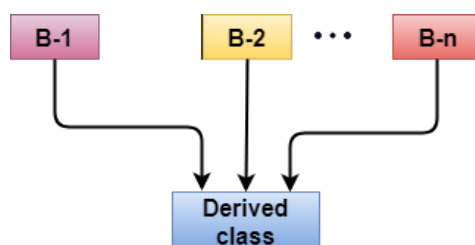
```

Eating...
Barking...
Weeping...

```

C++ Multiple Inheritance

Multiple inheritance is the process of deriving a new class that inherits the attributes from two or more classes.



Syntax of the Derived class:

```

1. class D : visibility B-1, visibility B-2, ?
2. {
3.     // Body of the class;
4. }

```

Let's see a simple example of multiple inheritance.

```

1. #include <iostream>
2. using namespace std;
3. class A

```

```
4. {
5.     protected:
6.         int a;
7.     public:
8.         void get_a(int n)
9.         {
10.             a = n;
11.         }
12. };
13.
14. class B
15. {
16.     protected:
17.         int b;
18.     public:
19.         void get_b(int n)
20.         {
21.             b = n;
22.         }
23. };
24. class C : public A,public B
25. {
26.     public:
27.         void display()
28.         {
29.             std::cout << "The value of a is : " <<a<< std::endl;
30.             std::cout << "The value of b is : " <<b<< std::endl;
31.             cout<<"Addition of a and b is : "<<a+b;
32.         }
33. };
34. int main()
35. {
36.     C c;
37.     c.get_a(10);
38.     c.get_b(20);
39.     c.display();
40. }
```

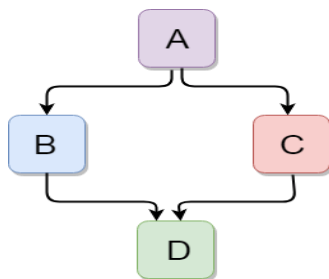
```
41. return 0;
42. }
```

Output:

```
The value of a is : 10
The value of b is : 20
Addition of a and b is : 30
```

C++ Hybrid Inheritance

Hybrid inheritance is a combination of more than one type of inheritance.



Let's see a simple example:

```
1. #include <iostream>
2. using namespace std;
3. class A
4. {
5.     protected:
6.         int a;
7.     public:
8.         void get_a()
9.         {
10.             std::cout << "Enter the value of 'a' : " << std::endl;
11.             cin >> a;
12.         }
13. };
14.
15. class B : public A
16. {
17.     protected:
18.         int b;
```

```

19. public:
20. void get_b()
21. {
22.     std::cout << "Enter the value of 'b' : " << std::endl;
23.     cin>>b;
24. }
25. };
26. class C
27. {
28. protected:
29. int c;
30. public:
31. void get_c()
32. {
33.     std::cout << "Enter the value of c is : " << std::endl;
34.     cin>>c;
35. }
36. };
37.
38. class D : public B, public C
39. {
40. protected:
41. int d;
42. public:
43. void mul()
44. {

45.     get_a();
46.     get_b();
47.     get_c();
48.     std::cout << "Multiplication of a,b,c is : " <<a*b*c<< std::endl;
49. }
50. };
51. int main()
52. {
53.     D d;

```



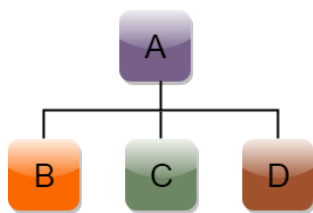
```
54. d.mul();
55. return 0;
56. }
```

Output:

```
Enter the value of 'a' :
10
Enter the value of 'b' :
20
Enter the value of c is :
30
Multiplication of a,b,c is : 6000
```

C++ Hierarchical Inheritance

Hierarchical inheritance is defined as the process of deriving more than one class from a base class.



Syntax of Hierarchical inheritance

```
1. class A
2. {
3.     // body of the class A.
4. }
5. class B : public A
6. {
7.     // body of class B.
8. }
9. class C : public A
10. {
11.     // body of class C.
12. }
13. class D : public A
14. {
15.     // body of class D.
```

16. }

Let's see a simple example:

```
1. #include <iostream>
2. using namespace std;
3. class Shape           // Declaration of base class.
4. {
5.     public:
6.     int a;
7.     int b;
8.     void get_data(int n,int m)
9.     {
10.         a= n;
11.         b = m;
12.     }
13.};
14. class Rectangle : public Shape // inheriting Shape class
15.{
16.     public:
17.     int rect_area()
18.     {
19.         int result = a*b;
20.         return result;
21.     }
22.};
23. class Triangle : public Shape // inheriting Shape class
24.{
25.     public:
26.     int triangle_area()
27.     {
28.         float result = 0.5*a*b;
29.         return result;
30.     }
31.};
32. int main()
33. {
```

```

34. Rectangle r;
35. Triangle t;
36. int length,breadth,base,height;
37. std::cout << "Enter the length and breadth of a rectangle: " << std::endl;
38. cin>>length>>breadth;
39. r.get_data(length,breadth);
40. int m = r.rect_area();
41. std::cout << "Area of the rectangle is : " <<m<< std::endl;
42. std::cout << "Enter the base and height of the triangle: " << std::endl;
43. cin>>base>>height;
44. t.get_data(base,height);
45. float n = t.triangle_area();
46. std::cout <<"Area of the triangle is : " << n<<std::endl;
47. return 0;
48.}

```

Output:

```

Enter the length and breadth of a rectangle:
23
20
Area of the rectangle is : 460
Enter the base and height of the triangle:
2
5
Area of the triangle is : 5

```

C++ Aggregation (HAS-A Relationship)

In C++, aggregation is a process in which one class defines another class as any entity reference. It is another way to reuse the class. It is a form of association that represents HAS-A relationship.

C++ Aggregation Example

Let's see an example of aggregation where Employee class has the reference of Address class as data member. In such way, it can reuse the members of Address class.

1. `#include <iostream>`
2. `using namespace std;`

```

3. class Address {
4.     public:
5.         string addressLine, city, state;
6.         Address(string addressLine, string city, string state)
7.         {
8.             this->addressLine = addressLine;
9.             this->city = city;
10.            this->state = state;
11.        }
12. };
13. class Employee
14. {
15.     private:
16.         Address* address; //Employee HAS-A Address
17.     public:
18.         int id;
19.         string name;
20.         Employee(int id, string name, Address* address)
21.         {
22.             this->id = id;
23.             this->name = name;
24.             this->address = address;
25.         }
26.         void display()
27.         {
28.             cout<<id <<" "<<name<<" "<<
29.             address->addressLine<<" "<< address->city<<" "<<address->state<<endl;
30.         }
31. };
32. int main(void) {
33.     Address a1= Address("C-146, Sec-15","Noida","UP");
34.     Employee e1 = Employee(101,"Nakul",&a1);
35.     e1.display();
36.     return 0;
37. }

```

Output:

CHAPTER-7

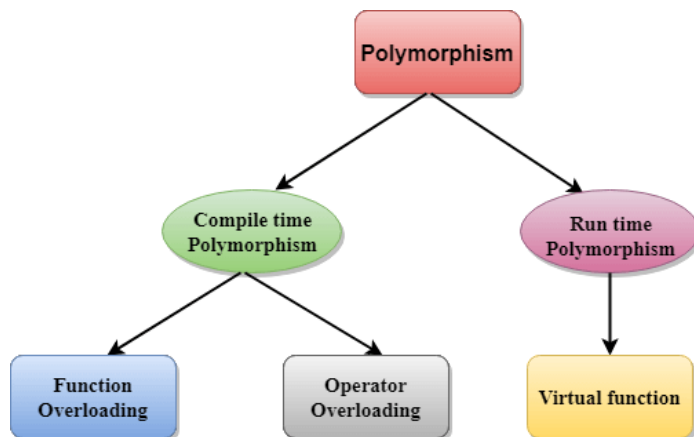
C++ Polymorphism

The term "Polymorphism" is the combination of "poly" + "morphs" which means many forms. It is a greek word. In object-oriented programming, we use 3 main concepts: inheritance, encapsulation, and polymorphism.

Real Life Example Of Polymorphism

Let's consider a real-life example of polymorphism. A lady behaves like a teacher in a classroom, mother or daughter in a home and customer in a market. Here, a single person is behaving differently according to the situations.

There are two types of polymorphism in C++:



- **Compile time polymorphism:** The overloaded functions are invoked by matching the type and number of arguments. This information is available at the compile time and, therefore, compiler selects the appropriate function at the compile time. It is achieved by function overloading and operator overloading which is also known as static binding or early binding. Now, let's consider the case where function name and prototype is same.

```
1. class A // base class declaration.
2. {
3.     int a;
4.     public:
5.     void display()
6.     {
```

```

7.         cout<< "Class A ";
8.     }
9. };
10. class B : public A           // derived class declaration.
11. {
12.     int b;
13.     public:
14.     void display()
15.     {
16.         cout<<"Class B";
17.     }
18. };

```

In the above case, the prototype of display() function is the same in both the **base and derived class**. Therefore, the static binding cannot be applied. It would be great if the appropriate function is selected at the run time. This is known as **run time polymorphism**.

- **Run time polymorphism:** Run time polymorphism is achieved when the object's method is invoked at the run time instead of compile time. It is achieved by method overriding which is also known as dynamic binding or late binding.

C++ Runtime Polymorphism Example

Let's see a simple example of run time polymorphism in C++.

// an example without the virtual keyword.

```

1. #include <iostream>
2. using namespace std;
3. class Animal {
4.     public:
5.     void eat(){
6.         cout<<"Eating...";
7.     }
8. };
9. class Dog: public Animal

```

```
10. {  
11. public:  
12. void eat()  
13. {      cout<<"Eating bread...";  
14. }  
15. };  
16. int main(void) {  
17.   Dog d = Dog();  
18.   d.eat();  
19.   return 0;  
20. }
```

Output:

```
Eating bread...
```

Differences b/w compile time and run time polymorphism.

Compile time polymorphism	Run time polymorphism
The function to be invoked is known at the compile time.	The function to be invoked is known at the run time.
It is also known as overloading, early binding and static binding.	It is also known as overriding, Dynamic binding and late binding.
Overloading is a compile time polymorphism where more than one method is having the same name but with the different number of parameters or the type of the parameters.	Overriding is a run time polymorphism where more than one method is having the same name, number of parameters and the type of the parameters.
It is achieved by function overloading and operator overloading.	It is achieved by virtual functions and pointers.
It provides fast execution as it is known at the compile time.	It provides slow execution as it is known at the run time.
It is less flexible as mainly all the things execute at the compile time.	It is more flexible as all the things execute at the run time.

C++ Overloading (Function and Operator)

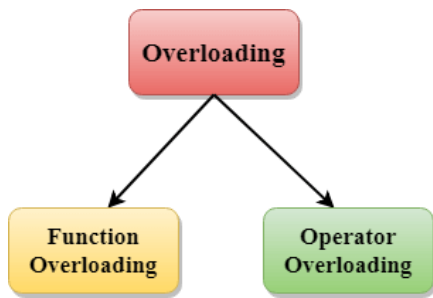
If we create two or more members having the same name but different in number or type of parameter, it is known as C++ overloading. In C++, we can overload:

- methods,
- constructors, and
- indexed properties

It is because these members have parameters only.

Types of overloading in C++ are:

- Function overloading
- Operator overloading



C++ Function Overloading

Function Overloading is defined as the process of having two or more function with the same name, but different in parameters is known as function overloading in C++. In function overloading, the function is redefined by using either different types of arguments or a different number of arguments. It is only through these differences compiler can differentiate between the functions.

The **advantage** of Function overloading is that it increases the readability of the program because you don't need to use different names for the same action.

C++ Function Overloading Example

Let's see the simple example of function overloading where we are changing number of arguments of add() method.

// program of function overloading when number of arguments vary.

```
1. #include <iostream>
2. using namespace std;
3. class Cal {
4.     public:
5.     static int add(int a,int b){
6.         return a + b;
7.     }
8.     static int add(int a, int b, int c)
9.     {
10.         return a + b + c;
11.     }
12. };
13. int main(void) {
14.     Cal C;                                // class object declaration.
```

```

15. cout<<C.add(10, 20)<<endl;
16. cout<<C.add(12, 20, 23);
17. return 0;
18. }

```

Output:

```

30
55

```

Let's see the simple example when the type of the arguments vary.

// Program of function overloading with different types of arguments.

```

1. #include<iostream>
2. using namespace std;
3. int mul(int,int);
4. float mul(float,int);
5.
6.
7. int mul(int a,int b)
8. {
9.     return a*b;
10. }
11. float mul(double x, int y)
12. {
13.     return x*y;
14. }
15. int main()
16. {
17.     int r1 = mul(6,7);
18.     float r2 = mul(0.2,3);
19.     std::cout << "r1 is : " <<r1<< std::endl;
20.     std::cout <<"r2 is : " <<r2<< std::endl;
21.     return 0;
22. }

```

Output:

```

r1 is : 42
r2 is : 0.6

```

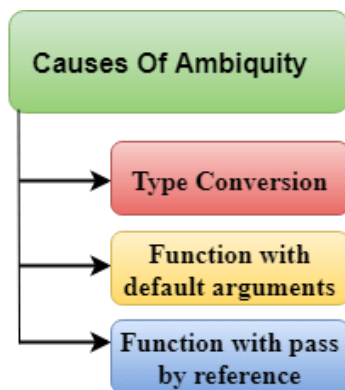
Function Overloading and Ambiguity

When the compiler is unable to decide which function is to be invoked among the overloaded function, this situation is known as **function overloading**.

When the compiler shows the ambiguity error, the compiler does not run the program.

Causes of Function Overloading:

- Type Conversion.
- Function with default arguments.
- Function with pass by reference.



- Type Conversion:

Let's see a simple example.

```
1. #include<iostream>
2. using namespace std;
3. void fun(int);
4. void fun(float);
5. void fun(int i)
6. {
7.     std::cout << "Value of i is : " << i << std::endl;
8. }
9. void fun(float j)
10. {
11.     std::cout << "Value of j is : " << j << std::endl;
12. }
13. int main()
```

```

14. {
15.   fun(12);
16.   fun(1.2);
17.   return 0;
18. }

```

The above example shows an error "**call of overloaded 'fun(double)' is ambiguous**". The fun(10) will call the first function. The fun(1.2) calls the second function according to our prediction. But, this does not refer to any function as in C++, all the floating point constants are treated as double not as a float. If we replace float to double, the program works. Therefore, this is a type conversion from float to double.

- Function with Default Arguments

Let's see a simple example.

```

1. #include<iostream>
2. using namespace std;
3. void fun(int);
4. void fun(int,int);
5. void fun(int i)
6. {
7.   std::cout << "Value of i is : " <<i<< std::endl;
8. }
9. void fun(int a,int b=9)
10. {
11.   std::cout << "Value of a is : " <<a<< std::endl;
12.   std::cout << "Value of b is : " <<b<< std::endl;
13. }
14. int main()
15. {
16.   fun(12);
17.
18.   return 0;
19. }

```

The above example shows an error "call of overloaded 'fun(int)' is ambiguous". The fun(int a, int b=9) can be called in two ways: first is by calling the function with one

argument, i.e., fun(12) and another way is calling the function with two arguments, i.e., fun(4,5). The fun(int i) function is invoked with one argument.

Therefore, the compiler could not be able to select among fun(int i) and fun(int a,int b=9).

- Function with pass by reference

Let's see a simple example.

```
1. #include <iostream>
2. using namespace std;
3. void fun(int);
4. void fun(int &);
5. int main()
6. {
7.     int a=10;
8.     fun(a); // error, which f()?
9.     return 0;
10. }
11. void fun(int x)
12. {
13.     std::cout << "Value of x is : " <<x<< std::endl;
14. }
15. void fun(int &b)
16. {
17.     std::cout << "Value of b is : " <<b<< std::endl;
18. }
```

The above example shows an error "**call of overloaded 'fun(int&)' is ambiguous**". The first function takes one integer argument and the second function takes a reference parameter as an argument. In this case, the compiler does not know which function is needed by the user as there is no syntactical difference between the fun(int) and fun(int &).

C++ Operators Overloading

Operator overloading is a compile-time polymorphism in which the operator is overloaded to provide the special meaning to the user-defined data type.

Operator overloading is used to overload or redefines most of the operators available in C++. It is used to perform the operation on the user-defined data type. For example, C++ provides the ability to add the variables of the user-defined data type that is applied to the built-in data types.

The advantage of Operators overloading is to perform different operations on the same operand.

Operator that cannot be overloaded are as follows:

- Scope operator (::)
- Sizeof
- member selector(.)
- member pointer selector(*)
- ternary operator(?:)

Syntax of Operator Overloading

1. `return_type class_name :: operator op(argument_list)`
2. `{`
3. `// body of the function.`
4. `}`

Where the **return type** is the type of value returned by the function.

class_name is the name of the class.

operator op is an operator function where op is the operator being overloaded, and the operator is the keyword.

Rules for Operator Overloading

- Existing operators can only be overloaded, but the new operators cannot be overloaded.
- The overloaded operator contains atleast one operand of the user-defined data type.
- We cannot use friend function to overload certain operators. However, the member function can be used to overload those operators.
- When unary operators are overloaded through a member function take no explicit arguments, but, if they are overloaded by a friend function, takes one argument.

- When binary operators are overloaded through a member function takes one explicit argument, and if they are overloaded through a friend function takes two explicit arguments.

C++ Operators Overloading Example

Let's see the simple example of operator overloading in C++. In this example, void operator ++ () operator function is defined (inside Test class).

// program to overload the unary operator ++.

```
1. #include <iostream>
2. using namespace std;
3. class Test
4. {
5.     private:
6.         int num;
7.     public:
8.         Test(): num(8){}
9.         void operator ++()    {
10.             num = num+2;
11.         }
12.         void Print() {
13.             cout<<"The Count is: "<<num;
14.         }
15. };
16. int main()
17. {
18.     Test tt;
19.     ++tt; // calling of a function "void operator ++()"
20.     tt.Print();

21.     return 0;
22. }
```

Output:

```
The Count is: 10
```


Let's see a simple example of overloading the binary operators.

// program to overload the binary operators.

```
1. #include <iostream>
2. using namespace std;
3. class A
4. {
5.     int x;
6.     public:
7.         A(){}
8.         A(int i)
9.         {
10.            x=i;
11.        }
12.        void operator+(A);
13.        void display();
14. };
15.
16. void A :: operator+(A a)
17. {
18.
19.     int m = x+a.x;
20.     cout<<"The result of the addition of two objects is : "<<m;
21.
22. }
23. int main()
24. {
25.     A a1(5);
26.     A a2(4);
27.     a1+a2;
28.     return 0;
29. }
```

Output:

```
The result of the addition of two objects is : 9
```

C++ Function Overriding

If derived class defines same function as defined in its base class, it is known as function overriding in C++. It is used to achieve runtime polymorphism. It enables you to provide specific implementation of the function which is already provided by its base class.

C++ Function Overriding Example

Let's see a simple example of Function overriding in C++. In this example, we are overriding the eat() function.

```
1. #include <iostream>
2. using namespace std;
3. class Animal {
4.     public:
5.     void eat(){
6.         cout<<"Eating...";
7.     }
8. };
9. class Dog: public Animal
10. {
11.     public:
12.     void eat()
13.     {
14.         cout<<"Eating bread...";
15.     }
16. };
17. int main(void) {
18.     Dog d = Dog();
19.     d.eat();
20.     return 0;
21. }
```

Output:

```
Eating bread...
```

C++ virtual function

- A C++ virtual function is a member function in the base class that you redefine in a derived class. It is declared using the virtual keyword.
- It is used to tell the compiler to perform dynamic linkage or late binding on the function.
- There is a necessity to use the single pointer to refer to all the objects of the different classes. So, we create the pointer to the base class that refers to all the derived objects. But, when base class pointer contains the address of the derived class object, always executes the base class function. This issue can only be resolved by using the 'virtual' function.
- A 'virtual' is a keyword preceding the normal declaration of a function.
- When the function is made virtual, C++ determines which function is to be invoked at the runtime based on the type of the object pointed by the base class pointer.

Late binding or Dynamic linkage

In late binding function call is resolved during runtime. Therefore compiler determines the type of object at runtime, and then binds the function call.

Rules of Virtual Function

- Virtual functions must be members of some class.
- Virtual functions cannot be static members.
- They are accessed through object pointers.
- They can be a friend of another class.
- A virtual function must be defined in the base class, even though it is not used.
- The prototypes of a virtual function of the base class and all the derived classes must be identical. If the two functions with the same name but different prototypes, C++ will consider them as the overloaded functions.
- We cannot have a virtual constructor, but we can have a virtual destructor
- Consider the situation when we don't use the virtual keyword.

1. `#include <iostream>`
2. `using namespace std;`
3. `class A`

```

4. {
5.     int x=5;
6.     public:
7.     void display()
8.     {
9.         std::cout << "Value of x is : " << x<<std::endl;
10.    }
11. };
12. class B: public A
13. {
14.     int y = 10;
15.     public:
16.     void display()
17.     {
18.         std::cout << "Value of y is : " <<y<< std::endl;
19.     }
20. };
21. int main()
22. {
23.     A *a;
24.     B b;
25.     a = &b;
26.     a->display();
27.     return 0;
28. }

```

Output:

```
Value of x is : 5
```

Interfaces in C++ (Abstract Classes)

Abstract classes are the way to achieve abstraction in C++. Abstraction in C++ is the process to hide the internal details and showing functionality only. Abstraction can be achieved by two ways:

1. Abstract class

2. Interface

Abstract class and interface both can have abstract methods which are necessary for abstraction.

C++ Abstract class

In C++ class is made abstract by declaring at least one of its functions as `<>strong>pure virtual function`. A pure virtual function is specified by placing `"= 0"` in its declaration. Its implementation must be provided by derived classes.

Let's see an example of abstract class in C++ which has one abstract method `draw()`. Its implementation is provided by derived classes: `Rectangle` and `Circle`. Both classes have different implementation.

```
1. #include <iostream>
2. using namespace std;
3. class Shape
4. {
5.     public:
6.     virtual void draw()=0;
7. };
8. class Rectangle : Shape
9. {
10.    public:
11.    void draw()
12.    {
13.        cout << "drawing rectangle..." << endl;
14.    }
15.};
16. class Circle : Shape
17. {
18.    public:
19.    void draw()
20.    {
21.        cout << "drawing circle..." << endl;
22.    }
23.};
24. int main() {
```

```
25. Rectangle rec;  
26. Circle cir;  
27. rec.draw();  
28. cir.draw();  
29. return 0;  
30.}
```

Output:

```
drawing rectangle...  
drawing circle...
```

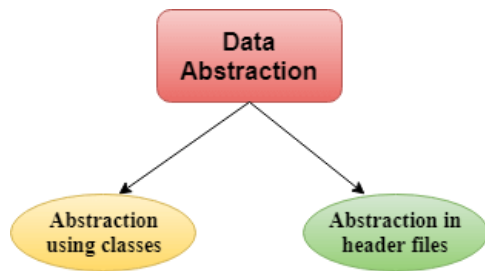
Data Abstraction in C++

- Data Abstraction is a process of providing only the essential details to the outside world and hiding the internal details, i.e., representing only the essential details in the program.
- Data Abstraction is a programming technique that depends on the separation of the interface and implementation details of the program.
- Let's take a real life example of AC, which can be turned ON or OFF, change the temperature, change the mode, and other external components such as fan, swing. But, we don't know the internal details of the AC, i.e., how it works internally. Thus, we can say that AC separates the implementation details from the external interface.
- C++ provides a great level of abstraction. For example, `pow()` function is used to calculate the power of a number without knowing the algorithm the function follows.

In C++ program if we implement class with private and public members then it is an example of data abstraction.

Data Abstraction can be achieved in two ways:

- Abstraction using classes
- Abstraction in header files.



Abstraction using classes: An abstraction can be achieved using classes. A class is used to group all the data members and member functions into a single unit by using the access specifiers. A class has the responsibility to determine which data member is to be visible outside and which is not.

Abstraction in header files: Another type of abstraction is header file. For example, `pow()` function available is used to calculate the power of a number without actually knowing which algorithm function uses to calculate the power. Thus, we can say that header files hide all the implementation details from the user.

Access Specifiers Implement Abstraction:

- **Public specifier:** When the members are declared as public, members can be accessed anywhere from the program.
- **Private specifier:** When the members are declared as private, members can only be accessed only by the member functions of the class.

Let's see a simple example of abstraction in header files.

// program to calculate the power of a number.

```
1. #include <iostream>
2. #include<math.h>
3. using namespace std;
4. int main()
5. {
6.     int n = 4;
7.     int power = 3;
8.     int result = pow(n,power);    // pow(n,power) is the power function
9.     std::cout << "Cube of n is : " << result<< std::endl;
10.    return 0;
11. }
```

Output:

```
Cube of n is : 64
```

Advantages Of Abstraction:

- Implementation details of the class are protected from the inadvertent user level errors.
- A programmer does not need to write the low level code.
- Data Abstraction avoids the code duplication, i.e., programmer does not have to undergo the same tasks every time to perform the similar operation.
- The main aim of the data abstraction is to reuse the code and the proper partitioning of the code across the classes.
- Internal implementation can be changed without affecting the user level code.

CHAPTER-8

C++ Strings

In C++, string is an object of **std::string** class that represents sequence of characters. We can perform many operations on strings such as concatenation, comparison, conversion etc.

C++ String Example

Let's see the simple example of C++ string.

```
#include <iostream>
using namespace std;
int main() {
    string s1 = "Hello";
    char ch[] = { 'C', '+', '+' };
    string s2 = string(ch);
    cout<<s1<<endl;
    cout<<s2<<endl;
}
```

Output:

```
Hello
C++
```