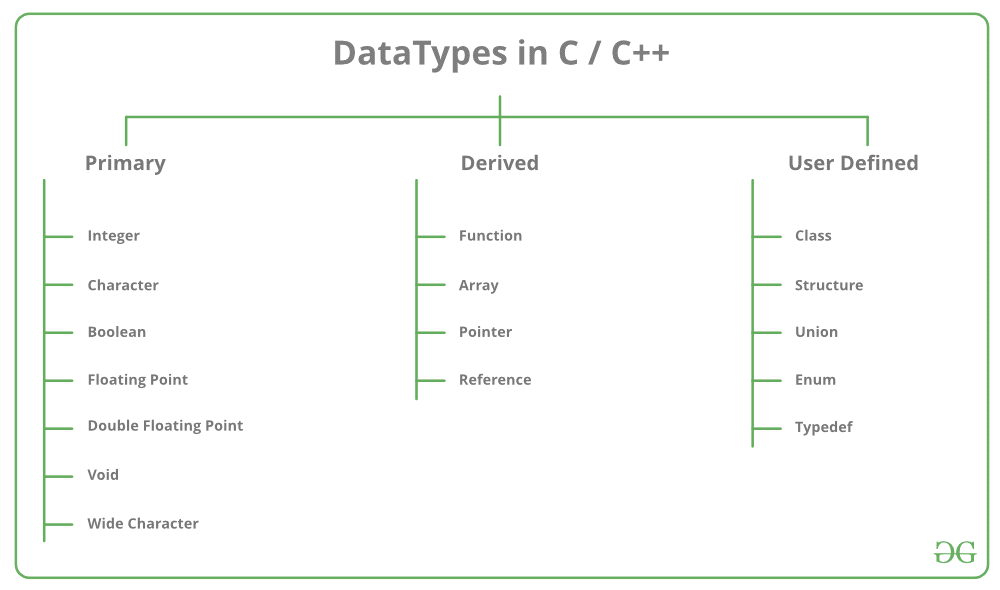
**C++ Data Types**

**All**[**variables**](https://www.geeksforgeeks.org/variables-and-keywords-in-c/)**use data-type during declaration to restrict the type of data to be stored. Therefore, we can say that data types are used to tell the variables the type of data it can store. Whenever a variable is defined in C++, the compiler allocates some memory for that variable based on the data-type with which it is declared. Every data type requires a different amount of memory.**



Data types in C++ is mainly divided into three types:

* **Primitive Data Types**: These data types are built-in or predefined data types and can be used directly by the user to declare variables. example: int, char , float, bool etc. Primitive data types available in C++ are:
* Integer
* Character
* Boolean
* Floating Point
* Double Floating Point
* Valueless or Void
* Wide Character
* [**Derived Data Types:**](https://www.geeksforgeeks.org/derived-data-types-in-c/) The data-types that are derived from the primitive or built-in datatypes are referred to as Derived Data Types. These can be of four types namely:
* Function
* Array
* Pointer
* Reference
* [**Abstract or User-Defined Data Types**](https://www.geeksforgeeks.org/user-defined-derived-data-types-in-c/)**:** These data types are defined by user itself. Like, defining a class in C++ or a structure. C++ provides the following user-defined datatypes:
* Class
* Structure
* Union
* Enumeration
* Typedef defined DataType

**This article discusses primitive data types available in C++.**

* **Integer**: Keyword used for integer data types is int. Integers typically requires 4 bytes of memory space and ranges from -2147483648 to 2147483647.
* **Character**: Character data type is used for storing characters. Keyword used for character data type is char. Characters typically requires 1 byte of memory space and ranges from -128 to 127 or 0 to 255.
* **Boolean**: Boolean data type is used for storing boolean or logical values. A boolean variable can store either *true*or *false*. Keyword used for boolean data type is bool.
* **Floating Point**: Floating Point data type is used for storing single precision floating point values or decimal values. Keyword used for floating point data type is float. Float variables typically requires 4 byte of memory space.
* **Double Floating Point**: Double Floating Point data type is used for storing double precision floating point values or decimal values. Keyword used for double floating point data type is double. Double variables typically requires 8 byte of memory space.
* **void**: Void means without any value. void datatype represents a valueless entity. Void data type is used for those function which does not returns a value.
* [Wide Character](https://www.geeksforgeeks.org/wide-char-and-library-functions-in-c/): Wide character data type is also a character data type but this data type has size greater than the normal 8-bit datatype. Represented by wchar\_t. It is generally 2 or 4 bytes long.

**Use of #include<iostream.h>**

**We use iostream because we use input output in program**

**And there are 2 classes which are predefined are given below-**

**Istreaminputcin>>**

**Ostreamoutputcout<<**

**Cin-console input**

**Cout-console output**

**Simple if statement**

**Syntax-if(condition)**

**{**

**Statement\_\_\_\_\_\_\_\_\_\_\_\_**

**\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_statement**

**}**

***In if statement if condition is true then statement will execute and if the condition is false then the statement will skip.***

***Program-***

**#include<iostream>**

**using namespace std;**

**int main()**

**{**

**int sal,b;**

**cout<<"enter your salary";**

**cin>>sal;**

**if (sal>=10000)**

**{**

**b=(sal\*10/100);**

**cout<<"bonus:"<<b;**

**sal=sal+b;**

**}**

**cout<<"salary after applying bonus"<<b;**

**return 0;**

**}**

***if else statement***

***Syntax-***

***If(condition)***

***{***

***Statement\_\_\_\_\_\_\_\_\_\_\_\_\_\_***

***\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_statement;***

***}***

***Else***

***{***

***Statement\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_***

***\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_statement;***

***}***

***In if else statement if the condition is not true then it execute its second block of else statement. but if the first condition is true then it not execute second statement.***

***#include<iostream>***

***using namespace std;***

***int main()***

***{***

***int a,b;***

***cout<<"enter your number:"<<a;***

***cin>>a;***

***if((a%2)==0)***

***{***

***cout<<"even number"<<a;***

***}***

***else***

***{***

***cout<<"odd number"<<a;***

***}***

***return 0;***

***}***

**Nested if statement**  
A nested if in C is an if statement that is the target of another if statement. Nested if statements means an if statement inside another if statement. Yes, both C and C++ allows us to nested if statements within if statements, i.e, we can place an if statement inside another if statement.  
**Syntax:**

if (condition1)



{

// Executes when condition1 is true

if (condition2)

{

// Executes when condition2 is true

}

}

Example-



#include <iostream>

using namespace std;

int main() {

// Declaring 3d array

int arr[3][3][3];

// Initializing the array



for (int i = 0; i < 3; i++) {

for (int j = 0; j < 3; j++) {

for (int k = 0; k < 3; k++) {

arr[i][j][k] = i + j + k;

}

}

}



// Printing the array

for (int i = 0; i < 3; i++) {

cout << i << "st layer:" << endl;

for (int j = 0; j < 3; j++) {

for (int k = 0; k < 3; k++) {

cout << arr[i][j][k] << " ";

}

cout << endl;

}

cout << endl;

}



return 0;

}



|  |
| --- |
| // C++ program to illustrate nested-if statement  #include <iostream>  using namespace std;    int main()  {      int i = 10;        if (i == 10)      {          // First if statement          if (i < 15)             cout<<"i is smaller than 15\n";            // Nested - if statement          // Will only be executed if statement above          // is true          if (i < 12)              cout<<"i is smaller than 12 too\n";          else              cout<<"i is greater than 15";      }        return 0;  } |

**Output:**

i is smaller than 15

i is smaller than 12 too

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

***While loop***

**#include <iostream>**

**using namespace std;**

**int main() {**

**int i = 0;**

**while (i < 5) {**

**cout << i << "\n";**

**i++;**

**}**

**return 0;**

**}**

* The while loop evaluates the test expression.
* If the test expression is true, codes inside the body of while loop is evaluated.
* Then, the test expression is evaluated again. This process goes on until the test expression is false.
* When the test expression is false, while loop is terminated.

*Do while loop*

// C++ Program to compute factorial of a number

// Factorial of n = 1\*2\*3...\*n

#include <iostream>

using namespace std;

int main()

{

int number, i = 1, factorial = 1;

cout << "Enter a positive integer: ";

cin >> number;

while ( i <= number) {

factorial \*= i; //factorial = factorial \* i;

++i;

}

cout<<"Factorial of "<< number <<" = "<< factorial;

return 0;

}

**Output**

Enter a positive integer: 4

Factorial of 4 = 24

In this program, user is asked to enter a positive integer which is stored in variable number. Let's suppose, user entered 4.

Then, the while loop starts executing the code. Here's how while loop works:

* Initially, i = 1, test expression i <= number is true and factorial becomes 1.
* Variable i is updated to 2, test expression is true, factorial becomes 2.
* Variable i is updated to 3, test expression is true, factorial becomes 6.
* Variable i is updated to 4, test expression is true, factorial becomes 24.
* Variable i is updated to 5, test expression is false and while loop is terminated.

**C++ do...while Loop**

The do...while loop is a variant of the while loop with one important difference. The body of do...while loop is executed once before the test expression is checked.

The syntax of do..while loop is:

do {

// codes;

}

while (testExpression);

**How do...while loop works?**

* The codes inside the body of loop is executed at least once. Then, only the test expression is checked.
* If the test expression is true, the body of loop is executed. This process continues until the test expression becomes false.
* When the test expression is false, do...while loop is terminated.
* **Example 2: C++ do...while Loop**
* // C++ program to add numbers until user enters 0
* #include <iostream>
* using namespace std;
* int main()
* {
* float number, sum = 0.0;
* do {
* cout<<"Enter a number: ";
* cin>>number;
* sum += number;
* }
* while(number != 0.0);
* cout<<"Total sum = "<<sum;
* return 0;
* }
* **Output**
* Enter a number: 2
* Enter a number: 3
* Enter a number: 4
* Enter a number: -4
* Enter a number: 2
* Enter a number: 4.4
* Enter a number: 2
* Enter a number: 0

**C++ for Loop**

*Loops are used in programming to repeat a specific block of code. In this tutorial, you will learn to create a for loop in C++ programming (with examples).*

Loops are used in programming to repeat a specific block until some end condition is met. There are three type of loops in C++ programming:

* for loop
* [while loop](https://www.programiz.com/cpp-programming/do-while-loop)
* [do...while loop](https://www.programiz.com/cpp-programming/do-while-loop)

**C++ for Loop Syntax**

for(initializationStatement; testExpression; updateStatement) {

// codes

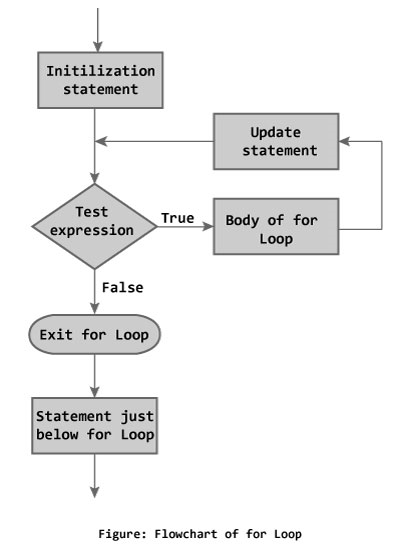
}

where, only testExpression is mandatory.

**How for loop works?**

* The initialization statement is executed only once at the beginning.
* Then, the test expression is evaluated.
* If the test expression is false, for loop is terminated. But if the test expression is true, codes inside body of for loop is executed and update expression is updated.
* Again, the test expression is evaluated and this process repeats until the test expression is false.

**Flowchart of for Loop in C++**



**Example 1: C++ for Loop**

// C++ Program to find factorial of a number

// Factorial on n = 1\*2\*3\*...\*n

#include <iostream>

using namespace std;

int main()

{

int i, n, factorial = 1;

cout << "Enter a positive integer: ";

cin >> n;

for (i = 1; i <= n; ++i) {

factorial \*= i; // factorial = factorial \* i;

}

cout<< "Factorial of "<<n<<" = "<<factorial;

return 0;

}

**Output**

Enter a positive integer: 5

Factorial of 5 = 120

In the program, user is asked to enter a positive integer which is stored in variable n (suppose user entered 5). Here is the working of for loop:

* Initially, i is equal to 1, test expression is true, factorial becomes 1.
* i is updated to 2, test expression is true, factorial becomes 2.
* i is updated to 3, test expression is true, factorial becomes 6.
* i is updated to 4, test expression is true, factorial becomes 24.
* i is updated to 5, test expression is true, factorial becomes 120.
* i is updated to 6, test expression is false, for loop is terminated.

In the above program, variable i is not used outside of the for loop. In such cases, it is better to declare the variable in for loop (at initialization statement).

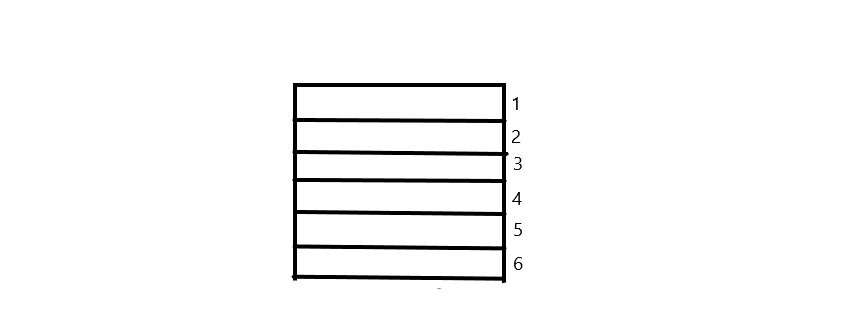
**Array in c++**

Array is the collection of same type of data type it also allow us to arrange data in matrix form and print number at exact location as you want.

Ther are two type of array in c++

* Single dimensional array
* Multi dimensional array
* Single dimensional array-syntax=type name[rows]

It store value in computer like this-

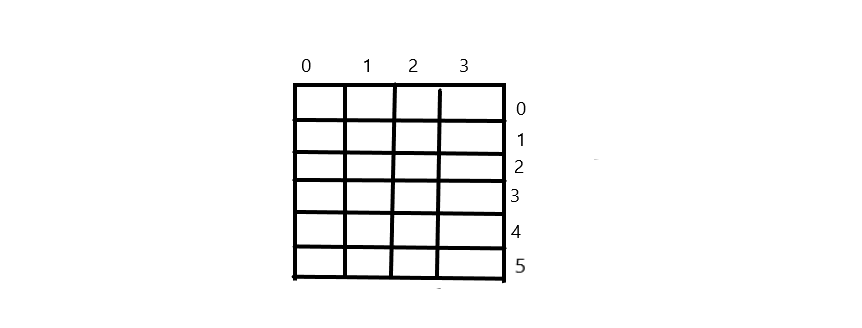


to initialize value while creating array do this-

Int no[5]{1,2,3,4,5};

Syntax-type name[rows][colums]

It store value in computer like this-



Definition-it allow to store value in matrices form

**FUNCTION**

A function is a block of code which only runs when it is called.

**Example Explained**

* myFunction() is the name of the function
* void means that the function does not have a return value. You will learn more about return values later in the next chapter
* inside the function (the body), add code that defines what the function should do

**types of function are-**

* built in function
* user defined function
* **Syntax for using Functions in C++**
* Here is how you define a function in C++,
* return-type function-name(parameter1, parameter2, ...)
* {
* // function-body
* }
* **A simple function example**
* #include <iostream>
* using namespace std;
* /\* This function adds two integer values
* \* and returns the result
* \*/int
* sum(int num1, int num2){
* int num3 = num1+num2; return num3;
* }
* int main(){
* //Calling the function
* cout<<sum(1,99);
* return 0;
* }
* **Output:**
* 100

**1) Build-it functions**

Built-in functions are also known as library functions. We need not to declare and define these functions as they are already written in the C++ libraries such as iostream, cmath etc. We can directly call them when we need.

***Example: C++ built-in function example***

Here we are using built-in function pow(x,y) which is x to the power y. This function is declared in cmath header file so we have included the file in our program using #include directive.

#include <iostream>

#include <cmath>

using namespace std;

int main(){

/\* Calling the built-in function

\* pow(x, y) which is x to the power y

\* We are directly calling this function

\*/

cout<<pow(2,5);

return 0;

}

**Output:**

32

**2) User-defined functions**

We have already seen user-defined functions, the example we have given at the beginning of this tutorial is an example of user-defined function. The functions that we declare and write in our programs are user-defined functions. Lets see another example of user-defined functions.

***User-defined functions***

#include <iostream>

#include <cmath>

using namespace std;

//Declaring the function sum

int sum(int,int);

int main(){

int x, y;

cout<<"enter first number: ";

cin>> x;

cout<<"enter second number: ";

cin>>y;

cout<<"Sum of these two :"<<sum(x,y);

return 0;

}

//Defining the function sum

int sum(int a, int b) {

int c = a+b;

return c;

}

**Output:**

enter first number: 22

enter second number: 19

Sum of these two :41

**User defined function type**

* Function wih argument and no return value
* Function with no argument and return value
* Function with no argument and no return value
* Function with argument and return value

Function wih argument and no return value

Syntax-void func(int,int);

**Int main()**

**{**

**Func(x,y);**

**}**

**Defi-void func(int?,int?)**

**{**

**Return ?;**

**}**

**Function with no argument and no return value**

**Prototype-void func(void);**

**Call-func();**

**Defi- void func(void)**

**{**

**}**

* **Function with no argument and return value**

**Prototype-int func(void)**

**Call=variable=func();**

**Defi-int func(void)**

**{**

**}**

**Function with argument and return value**

**Prototype-int func(int);**

**Call=variable=func(variable declared in main function)**

**Defi-int func(int ?)**

**{**

**Return ?**

**}**

There are three ways to make function in c++

* FUNCTION PROTOTYPE
* FUNCTION DEFINATION
* FUNCTION CALL

Function prototype-

Function type

* function name-return value type
* Parameter list-what type of argument recieve
* Semi colon-;to end code
* Syntax-int

Function definition

Function header

Function body

Type functionname(parameter list)

{

Local variable declaration//;

--------------------statement--------------

-----------------------------------------------;

Return statement;

}

**Call by value and call by address**

In call by address variable address pass through main function where as in call by value variable copy passes through main function.

**Syntax-**

Prototype-void func(int\*,int\*);

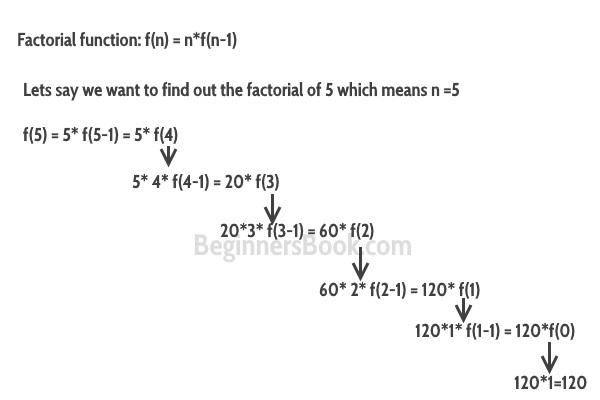
call-func(&?,&?)

defi-void func(\*?,\*?)

The process in which a function calls itself is known as recursion and the corresponding function is called the **recursive function**. The popular example to understand the recursion is factorial function.

**Factorial function:** f(n) = n\*f(n-1), base condition: if n<=1 then f(n) = 1. Don’t worry we wil discuss what is base condition and why it is important.

In the following diagram. I have shown that how the factorial function is calling itself until the function reaches to the base condition.



Lets solve the problem using C++ program.

**C++ recursion example: Factorial**

#include <iostream>

using namespace std;

//Factorial function

int f(int n){

/\* This is called the base condition, it is

\* very important to specify the base condition

\* in recursion, otherwise your program will throw

\* stack overflow error.

\*/

   if (n <= 1)

        return 1;

   else

      return n\*f(n-1);

}

int main(){

int num;

  cout<<"Enter a number: ";

   cin>>num;

   cout<<"Factorial of entered number: "<<f(num);

return 0;

}

**Output:**

Enter a number: 5

Factorial of entered number: 120

**Base condition**

In the above program, you can see that I have provided a base condition in the recursive function. The condition is:

if (n <= 1)

        return 1;

The purpose of recursion is to divide the problem into smaller problems till the base condition is reached. For example in the above factorial program I am solving the factorial function f(n) by calling a smaller factorial function f(n-1), this happens repeatedly until the n value reaches base condition(f(1)=1). If you do not define the base condition in the recursive function then you will get stack overflow error.

**Direct recursion vs indirect recursion**

**Direct recursion:** When function calls itself, it is called direct recursion, the example we have seen above is a direct recursion example.

**Indirect recursion:** When function calls another function and that function calls the calling function, then this is called indirect recursion. For example: function A calls function B and Function B calls function A.

**Indirect Recursion Example in C++**

#include <iostream>

using namespace std;

int fa(int);

int fb(int);

int fa(int n){

if(n<=1)

return 1;

else

return n\*fb(n-1);

}

int fb(int n){

if(n<=1)

return 1;

else

return n\*fa(n-1);

}

int main(){

int num=5;

   cout<<fa(num);

return 0;

}

Output:

120

**Function overloading in C++**

BY CHAITANYA SINGH | FILED UNDER: [LEARN C++](https://beginnersbook.com/category/learn-c/)

Function overloading is a [C++ programming](https://beginnersbook.com/2017/08/c-plus-plus-tutorial-for-beginners/) feature that allows us to have more than one function having same name but different parameter list, when I say parameter list, it means the data type and sequence of the parameters, for example the parameters list of a function myfuncn(int a, float b) is (int, float) which is different from the function myfuncn(float a, int b) parameter list (float, int). Function overloading is a [compile-time polymorphism](https://beginnersbook.com/2017/08/cpp-polymorphism/).  
Now that we know what is parameter list lets see the rules of overloading: we can have following functions in the same

#include *<iostream>*

**using** **namespace** **std**;

void add(int a, int b)

{

cout << "sum = " << (a + b);

}

void add(double a, double b)

{

cout << endl << "sum = " << (a + b);

}

*// Driver code*

int main()

{

add(10, 2);

add(5.3, 6.2);

**return** 0;

}

C++ inline function

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

Any change to an inline function could require all clients of the function to be recompiled because compiler would need to replace all the code once again otherwise it will continue with old functionality.

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

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

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

#include <iostream>

using namespace std;

inline int Max(int x, int y) {

return (x > y)? x : y;

}

// Main function for the program

int main() {

cout << "Max (20,10): " << Max(20,10) << endl;

cout << "Max (0,200): " << Max(0,200) << endl;

cout << "Max (100,1010): " << Max(100,1010) << endl;

return 0;

}

# **DEFAULT Argument**

A default argument is a value provided in a function declaration that is automatically assigned by the compiler if the caller of the function doesn’t provide a value for the argument with a default value.

Following is a simple C++ example to demonstrate the use of default arguments. We don’t have to write 3 sum functions, only one function works by using default values for 3rd and 4th arguments.

example

|  |
| --- |
| #include<iostream>  using namespace std;    // A function with default arguments, it can be called with  // 2 arguments or 3 arguments or 4 arguments.  int sum(int x, int y, int z=0, int w=0)  {      return (x + y + z + w);  }    /\* Driver program to test above function\*/  int main()  {      cout << sum(10, 15) << endl;      cout << sum(10, 15, 25) << endl;      cout << sum(10, 15, 25, 30) << endl;      return 0;  } |

Output

25

50

80

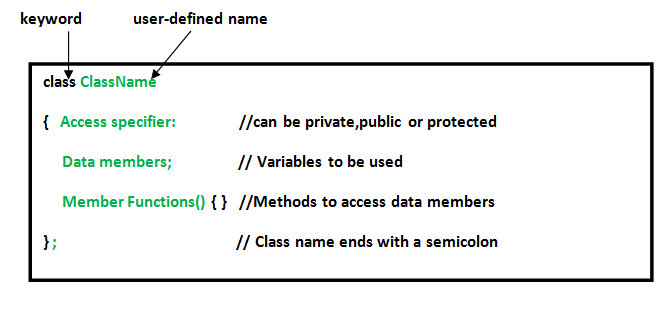
# **CONST ARGUMENTS**

In C++, an argument to a function can be declared as const. The argument with constant value should be initialized during the function declaration.  
**Syntax**  
*type function\_name(const data\_type variable\_name=value);*  
**For example**  
*int max(const int a=3, int b);         //function prototype*

**Classes and object in c++**

**Class:** A class in C++ is the building block, that leads to Object-Oriented programming. It is a user-defined data type, which holds its own data members and member functions, which can be accessed and used by creating an instance of that class. A C++ class is like a blueprint for an object.

**Defining class and object**

A class is defined in C++ using keyword class followed by the name of class. The body of class is defined inside the curly brackets and terminated by a semicolon at the end.

**Declaring Objects:** When a class is defined, only the specification for the object is defined; no memory or storage is allocated. To use the data and access functions defined in the class, you need to create objects.

**Syntax:**

**ClassName ObjectName;**

**Syntax-**Object name.function name(parameter list);

Visibility level of class in c++ are of three types =private,public,protected

Accessing specifier in c++

**Private**- in private data is written can only be access by class member they cannot be accessed by object directly but can be accessed through class function.if can t use any keyword while creating class it can store by default in private.

**Public-**in public code is free to access by object directly mostly class function is wrote under public because of security purposes .

**Protected-**it is lie between private and public it has restriction like private but it can also access by derived class it is mainly used for inheritance.

Syntax-class test

{

Private:

Protected:

Public:

};

We can summarize the different access types according to - who can access them in the following way −

|  |  |  |  |
| --- | --- | --- | --- |
| **Access** | **public** | **protected** | **private** |
| Same class | yes | yes | yes |
| Derived classes | yes | yes | no |
| Outside classes | yes | no | no |

Defining member function

1)inside the class definition-to define a member inside a class you can simply write it definition and body like in simple c++ program.

Syntax- Type functionname(parameter list)

{

Local variable declaration//;

--------------------statement--------------

-----------------------------------------------;

Return statement;

}

2)outside the class definition- To define a member function outside the class definition we have to use the scope resolution :: operator along with class name and function name.

Syntax=

Return type classname::function name(parameter lists)

{ |  
body |\_\_\_\_\_\_\_\_\_\_\_scope resolution operator   
}

A simple class and object program in c++

#include<iostream>

using namespace std;

class sum

{

int a,b,c;

public:

void getdata();

void putdata();

};

void sum::getdata(void)

{

cout<<"enter the value of a&b\n";

cin>>a>>b;

}

void sum::putdata(void)

{

c=a+b;

cout<<"sum of"<<a<<b<<"="<<c;

}

int main()

{

sum obj;

obj.getdata();

obj.putdata();

return 0;

}

Static data member

Static data members are class members that are declared using the static keyword. There is only one copy of the static data member in the class, even if there are many class objects. This is because all the objects share the static data member. The static data member is always initialized to zero when the first class object is created.

The syntax of the static data members is given as follows −

static data\_type data\_member\_name;

definition after writing class- data type classname::name = value;

#include <iostream>

#include<string.h>

using namespace std;

class Student {

private:

int rollNo;

char name[10];

int marks;

public:

static int objectCount;

Student() {

objectCount++;

}

void getdata() {

cout << "Enter roll number: "<<endl;

cin >> rollNo;

cout << "Enter name: "<<endl;

cin >> name;

cout << "Enter marks: "<<endl;

cin >> marks;

}

void putdata() {

cout<<"Roll Number = "<< rollNo <<endl;

cout<<"Name = "<< name <<endl;

cout<<"Marks = "<< marks <<endl;

cout<<endl;

}

};

int Student::objectCount = 0;

int main(void) {

Student s1;

s1.getdata();

s1.putdata();

Student s2;

s2.getdata();

s2.putdata();

Student s3;

s3.getdata();

s3.putdata();

cout << "Total objects created = " << Student::objectCount << endl;

return 0;

}

**The reason we need Static member function:**

* Static members are frequently used to store information that is shared by all objects in a class.
* For instance, you may keep track of the quantity of newly generated objects of a specific class type using a static data member as a counter. This static data member can be increased each time an object is generated to keep track of the overall number of objects.

**Example:**

* C++

|  |
| --- |
| // C++ Program to show the working of  // static member functions  #include <iostream>  **using** **namespace** std;    **class** Box  {  **private**:  **static** **int** length;  **static** **int** breadth;  **static** **int** height;    **public**:    **static** **void** print()      {          cout << "The value of the length is: " << length << endl;          cout << "The value of the breadth is: " << breadth << endl;          cout << "The value of the height is: " << height << endl;      }  };    // initialize the static data members    **int** Box :: length = 10;  **int** Box :: breadth = 20;  **int** Box :: height = 30;    // Driver Code    **int** main()  {        Box b;        cout << "Static member function is called through Object name: \n" << endl;      b.print();        cout << "\nStatic member function is called through Class name: \n" << endl;      Box::print();    **return** 0;  } |

**Output**

Static member function is called through Object name:

The value of the length is: 10

The value of the breadth is: 20

The value of the height is: 30

Static member function is called through Class name:

The value of the length is: 10

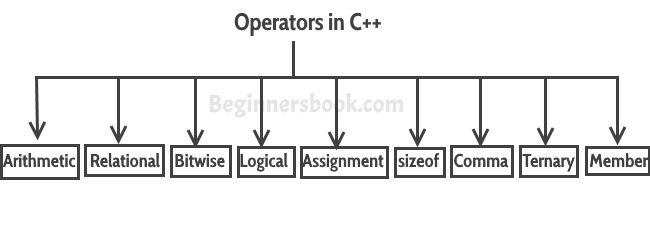
The value of the breadth is: 20

The value of the height is: 30

**Operators in C++**

Operator represents an action. For example + is an operator that represents addition. An operator works on two or more operands and produce an output. For example 3+4+5 here + operator works on three operands and produce 12 as output.

**Types of Operators in C++**



1) Basic Arithmetic Operators  
2) Assignment Operators  
3) Auto-increment and Auto-decrement Operators  
4) Logical Operators  
5) Comparison (relational) operators  
6) Bitwise Operators  
7) Ternary Operator

**1) Basic Arithmetic Operators**

Basic arithmetic operators are: +, -, \*, /, %  
**+** is for addition.

**–** is for subtraction.

**\*** is for multiplication.

**/** is for division.

**%** is for modulo.  
**Note**: Modulo operator returns remainder, for example 20 % 5 would return 0

**Example of Arithmetic Operators**

#include <iostream>

using namespace std;

int main(){

int num1 = 240;

 int num2 = 40;

cout<<"num1 + num2: "<<(num1 + num2)<<endl;

cout<<"num1 - num2: "<<(num1 - num2)<<endl;

 cout<<"num1 \* num2: "<<(num1 \* num2)<<endl;

 cout<<"num1 / num2: "<<(num1 / num2)<<endl;

cout<<"num1 % num2: "<<(num1 % num2)<<endl;

return 0;

}

**Output:**

num1 + num2: 280

num1 - num2: 200

num1 \* num2: 9600

num1 / num2: 6

num1 % num2: 0

**2) Assignment Operators**

Assignments operators in C++ are: =, +=, -=, \*=, /=, %=

**num2 = num1** would assign value of variable num1 to the variable.

**num2+=num1** is equal to num2 = num2+num1

**num2-=num1** is equal to num2 = num2-num1

**num2\*=num1** is equal to num2 = num2\*num1

**num2/=num1** is equal to num2 = num2/num1

**num2%=num1** is equal to num2 = num2%num1

**Example of Assignment Operators**

#include <iostream>

using namespace std;

int main(){

int num1 = 240;

int num2 = 40;

num2 = num1;

cout<<"= Output: "<<num2<<endl;

num2 += num1;

cout<<"+= Output: "<<num2<<endl;

num2 -= num1;

 cout<<"-= Output: "<<num2<<endl;

num2 \*= num1;

cout<<"\*= Output: "<<num2<<endl;

num2 /= num1;

cout<<"/= Output: "<<num2<<endl;

num2 %= num1;

cout<<"%= Output: "<<num2<<endl;

return 0;

}

**Output:**

= Output: 240

+= Output: 480

-= Output: 240

\*= Output: 57600

/= Output: 240

%= Output: 0

**3) Auto-increment and Auto-decrement Operators**

++ and —  
num++ is equivalent to num=num+1;

num–- is equivalent to num=num-1;

**Example of Auto-increment and Auto-decrement Operators**

#include <iostream>

using namespace std;

int main(){

int num1 = 240;

int num2 = 40;

num1++; num2--;

cout<<"num1++ is: "<<num1<<endl;

cout<<"num2-- is: "<<num2;

return 0;

}

**Output:**

num1++ is: 241

num2-- is: 39

**4) Logical Operators**

Logical Operators are used with binary variables. They are mainly used in conditional statements and loops for evaluating a condition.

Logical operators in C++ are: &&, ||, !

Let’s say we have two boolean variables b1 and b2.

**b1&&b2** will return true if both b1 and b2 are true else it would return false.

**b1||b2** will return false if both b1 and b2 are false else it would return true.

**!b1** would return the opposite of b1, that means it would be true if b1 is false and it would return false if b1 is true.

**Example of Logical Operators**

#include <iostream>

using namespace std;

int main(){

bool b1 = true;

   bool b2 = false;

cout<<"b1 && b2: "<<(b1&&b2)<<endl;

  cout<<"b1 || b2: "<<(b1||b2)<<endl;

   cout<<"!(b1 && b2): "<<!(b1&&b2);

return 0;

}

**Output:**

b1 && b2: 0

b1 || b2: 1

!(b1 && b2): 1

**5) Relational operators**

We have six relational operators in C++: ==, !=, >, <, >=, <=

**==** returns true if both the left side and right side are equal

**!=** returns true if left side is not equal to the right side of operator.

**>** returns true if left side is greater than right.

**<** returns true if left side is less than right side.

**>=** returns true if left side is greater than or equal to right side.

**<=** returns true if left side is less than or equal to right side.

**Example of Relational operators**

#include <iostream>

using namespace std;

int main(){

int num1 = 240;

int num2 =40;

if (num1==num2) {

cout<<"num1 and num2 are equal"<<endl;

   }

  else{

cout<<"num1 and num2 are not equal"<<endl;

  }

if( num1 != num2 ){

     cout<<"num1 and num2 are not equal"<<endl;

 }

 else{

     cout<<"num1 and num2 are equal"<<endl;

  }

if( num1 > num2 ){

     cout<<"num1 is greater than num2"<<endl;

  }

  else{

     cout<<"num1 is not greater than num2"<<endl;

  }

if( num1 >= num2 ){

      cout<<"num1 is greater than or equal to num2"<<endl;

  }

  else{

     cout<<"num1 is less than num2"<<endl;

  }

if( num1 < num2 ){

cout<<"num1 is less than num2"<<endl;

  }

  else{

cout<<"num1 is not less than num2"<<endl;

  }

if( num1 <= num2){

cout<<"num1 is less than or equal to num2"<<endl;

  }

  else{

cout<<"num1 is greater than num2"<<endl;

  }

return 0;

}

**Output:**

num1 and num2 are not equal

num1 and num2 are not equal

num1 is greater than num2

num1 is greater than or equal to num2

num1 is not less than num2

num1 is greater than num2

**7) Ternary Operator**

This operator evaluates a boolean expression and assign the value based on the result.  
Syntax:

variable num1 = (expression) ? value if true : value if false

If the expression results true then the first value before the colon (:) is assigned to the variable num1 else the second value is assigned to the num1.

**Example of Ternary Operator**

#include <iostream>

using namespace std;

int main(){

int num1, num2; num1 = 99;

/\* num1 is not equal to 10 that's why

\* the second value after colon is assigned

\* to the variable num2

\*/

num2 = (num1 == 10) ? 100: 200;

cout<<"num2: "<<num2<<endl;

/\* num1 is equal to 99 that's why

\* the first value is assigned

\* to the variable num2

\*/

num2 = (num1 == 99) ? 100: 200;

cout<<"num2: "<<num2;

return 0;

}

**Output:**

num2: 200

num2: 100

**Miscellaneous Operators**

There are few other operators in C++ such as **Comma operator** and **sizeof operator**. We will cover them in detail in a separate tutorial.

**Operator Precedence in C++**

This determines which operator needs to be evaluated first if an expression has more than one operator. Operator with higher precedence at the top and lower precedence at the bottom.

**Unary Operators**  
++ – – ! ~

**Multiplicative**  
\* / %

**Additive**  
+ –

**Shift**  
<< >> >>>

**Relational**  
> >= < <=

**Equality**  
== !=  
 **Bitwise AND**  
&  
**Bitwise XOR**  
^  
 **Bitwise OR**  
|

**Logical AND**  
&&

**Logical OR**  
||

**Ternary**  
?:

**Assignment**  
= += -= \*= /= %= > >= < <= &= ^= |=

# **Passing an Object as Function argument**

To pass an object as an argument we write the object name as the argument while calling the function the same way we do it for other variables.

**Syntax:**

Return type function\_name(object\_name as argument);

**#include<iostream>**

**using namespace std;**

**class time**

**{**

**int h,m;**

**public:**

**void gettime(int,int);**

**void puttime(void);**

**void sum(time,time);**

**};**

**void time::gettime(int x,int y)**

**{**

**h=x;**

**m=y;**

**}**

**void time::puttime(void)**

**{**

**cout<<"hour="<<h<<"\t minute="<<m;**

**}**

**void time::sum(time t1,time t2)**

**{**

**m=t1.m+t2.m;**

**h=m/60;**

**m=m%60;**

**h=h+t1.h+t2.h;**

**}**

**int main()**

**{**

**time t1,t2,t3;**

**t1.gettime(1,40);**

**t2.gettime(1,50);**

**t3.sum(t1,t2);**

**t1.puttime();**

**t2.puttime();**

**t3.puttime();**

**return 0;**

**}**

Static Members of a C++ Class

**A static member function is a special member function, which is used to access only static data members**, any other normal data member cannot be accessed through static member function. Just like static data member, static member function is also a class function; it is not associated with any class object

**Syntax-**

**Class name**

**{**

**Static void disp(parameter);**

**}**

**Void name::disp(parameter)**

**{**

**}**

**Int main**

**{**

class\_name:: function\_name(perameter);

**}**

#include<iostream>

using namespace std;

class name

{

int a;

static int count;

public:

void setno(void);

void dispno(void);

static void counter(void);

};

void name::setno(void)

{

a=++count;

}

void name::dispno(void)

{

cout<<"object no="<<a;

}

void name::counter(void)

{

cout<<"couter value="<<count;

}

int name::count;

int main()

{

name n1,n2,n3;

n1.setno();

n2.setno();

name::counter();

n1.dispno();

n2.dispno();

}

**The reason we need Static member function:**

* Static members are frequently used to store information that is shared by all objects in a class.
* For instance, you may keep track of the quantity of newly generated objects of a specific class type using a static data member as a counter. This static data member can be increased each time an object is generated to keep track of the overall number of objects.

**Example:**

* C++

|  |
| --- |
| // C++ Program to show the working of  // static member functions  #include <iostream>  **using** **namespace** std;    **class** Box  {  **private**:  **static** **int** length;  **static** **int** breadth;  **static** **int** height;    **public**:    **static** **void** print()      {          cout << "The value of the length is: " << length << endl;          cout << "The value of the breadth is: " << breadth << endl;          cout << "The value of the height is: " << height << endl;      }  };    // initialize the static data members    **int** Box :: length = 10;  **int** Box :: breadth = 20;  **int** Box :: height = 30;    // Driver Code    **int** main()  {        Box b;        cout << "Static member function is called through Object name: \n" << endl;      b.print();        cout << "\nStatic member function is called through Class name: \n" << endl;      Box::print();    **return** 0;  } |

**Output**

Static member function is called through Object name:

The value of the length is: 10

The value of the breadth is: 20

The value of the height is: 30

Static member function is called through Class name:

The value of the length is: 10

The value of the breadth is: 20

The value of the height is: 30

**Friend function**

**Rules while creating friend function**

**1-friend function is not the member function of class so it is not required to use classname:: while defining outside.**

**2-friend function call like a normal function in main program it does not need to use objectname while calling it.**

**3-void disp (class obj,class obj)**

**{**

**obj.variable;**

**obj.variable;**

**}**

**4-whie defining (friend return function(class1,class2);**

#include<iostream>

using namespace std;

class d;

class g

{

int a,b;

public:

void getdata(int x,int y)

{

a=x;

b=y;

}

void putdata(void)

{

cout<<"class a val="<<a<<"and"<<b;

}

friend void sum(g,d);

};

class d

{

int E,c;

public:

void getdata(int z,int V)

{

E=z;

c=V;

}

void putdata(void)

{

cout<<"class b val="<<E<<"and"<<c;

}

friend void sum(g,d);

};

void sum(g c1,d c2)

{

int t;

t=(c1.a+c1.b+c2.E+c2.c);

cout<<"sum of two object="<<t;

}

int main()

{

g A;

d B;

A.getdata(2,2);

B.getdata(4,4);

A.putdata();

B.putdata();

sum(A,B);

}

**Returning Object as argument**

**Syntax:**

object = return object\_name;

#include<iostream>

using namespace std;

class b;

class test

{

int a,b;

public:

void getval(int x,int y)

{

a=x;

b=y;

}

friend test sum(test,test);

void dispval(test);

};

test sum(t o1,test o2)

{

test o3;

o3.a=o1.a+o2.a;

o3.b=o1.b+o2.b;

return o3;

}

void test::dispval(test o3)

{

cout<<"sum 1="<<o3.a;

cout<<"sum 2="<<o3.b;

}

int main()

{

test o,b,c;

o.getval(3,3);

b.getval(4,3);

c=sum(o,b);

o.dispval(o);

b.dispval(b);

c.dispval(c);

return 0;

}

**scope resolution operator**

In C++, scope resolution operator is **::**. It is used for following purposes.

**1) To access a global variable when there is a local variable with same name:**

|  |
| --- |
| // C++ program to show that we can access a global variable  // using scope resolution operator :: when there is a local  // variable with same name  #include<iostream>  using namespace std;    int x;  // Global x    int main()  {    int x = 10; // Local x    cout << "Value of global x is " << ::x;    cout << "\nValue of local x is " << x;    return 0;  } |

Output:

Value of global x is 0

Value of local x is 10

**2) To define a function outside a class.**

|  |
| --- |
| // C++ program to show that scope resolution operator :: is used  // to define a function outside a class  #include<iostream>  using namespace std;    class A  {  public:       // Only declaration     void fun();  };    // Definition outside class using ::  void A::fun()  {     cout << "fun() called";  }    int main()  {     A a;     a.fun();     return 0;  } |

Output:

fun() called

**3) To access a class’s static variables.**

|  |
| --- |
| // C++ program to show that :: can be used to access static  // members when there is a local variable with same name  #include<iostream>  using namespace std;    class Test  {      static int x;  public:      static int y;        // Local parameter 'a' hides class member      // 'a', but we can access it using ::      void func(int x)      {         // We can access class's static variable         // even if there is a local variable         cout << "Value of static x is " << Test::x;           cout << "\nValue of local x is " << x;      }  };    // In C++, static members must be explicitly defined  // like this  int Test::x = 1;  int Test::y = 2;    int main()  {      Test obj;      int x = 3 ;      obj.func(x);        cout << "\nTest::y = " << Test::y;        return 0;  } |

Output:

Value of static x is 1

Value of local x is 3

Test::y = 2;

**4) In case of multiple Inheritance:**  
If same variable name exists in two ancestor classes, we can use scope resolution operator to distinguish.

|  |
| --- |
| // Use of scope resolution operator in multiple inheritance.  #include<iostream>  using namespace std;    class A  {  protected:      int x;  public:      A() { x = 10; }  };    class B  {  protected:      int x;  public:      B() { x = 20; }  };    class C: public A, public B  {  public:     void fun()     {        cout << "A's x is " << A::x;        cout << "\nB's x is " << B::x;     }  };    int main()  {      C c;      c.fun();      return 0;  } |

Output:

A's x is 10

B's x is 20

**5) For namespace**  
If a class having the same name exists inside two namespace we can use the namespace name with the scope resolution operator to refer that class without any conflicts

|  |
| --- |
| // Use of scope resolution operator for namespace.  #include<iostream>      int main(){      std::cout << "Hello" << std::endl;    } |

Here, cout and endl belong to the std namespace.

**6) Refer to a class inside another class:**  
If a class exists inside another class we can use the nesting class to refer the nested class using the scope resolution operator

|  |
| --- |
| // Use of scope resolution class inside another class.  #include<iostream>  using namespace std;    class outside  {  public:        int x;        class inside        {        public:              int x;              static int y;              int foo();          };  };  int outside::inside::y = 5;    int main(){      outside A;      outside::inside B;    }  \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ |

**constructor**

**constructor is a member function of a class which is used to initialize value of object .some other uses of constructor discussed in other lines.**

**Rules while making constructor are-**

* **Constructor should be made in public :**
* **Constructor does not return any value means it has no return value.**
* **Constructor is does not inherit in inheritance.**
* **Constructor should not be virtual.**
* **Constructor will not refer by his address**
* **Default argument can be used in constructor.**

**Syntax-**

**class test**

**{**

**Int a,b;**

**Public:  
test()**

**{**

**A=0;**

**B=0;**

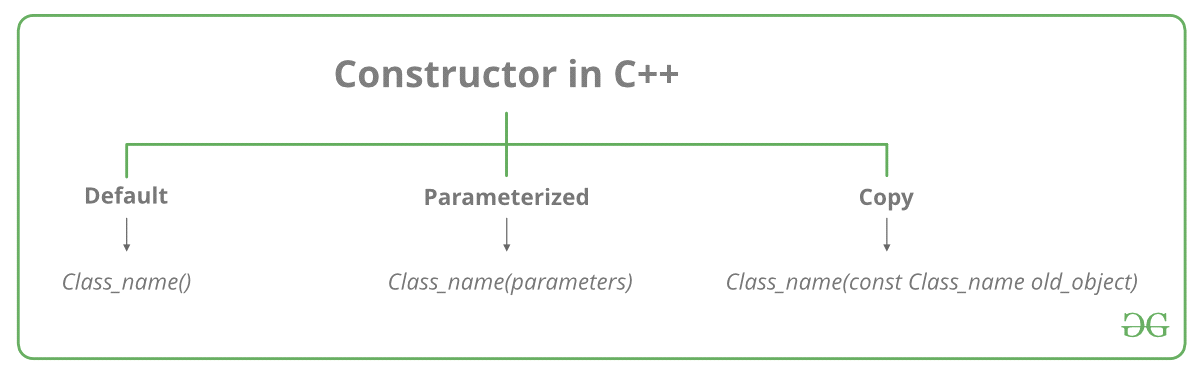
**}**

**};**

* Constructor has same name as the class itself
* Constructors don’t have return type
* A constructor is automatically called when an object is created.
* If we do not specify a constructor, C++ compiler generates a default constructor for us (expects no parameters and has an empty body).

A constructor is different from normal functions in following ways:

* Constructor has same name as the class itself
* Constructors don’t have return type
* A constructor is automatically called when an object is created.
* If we do not specify a constructor, C++ compiler generates a default constructor for us (expects no parameters and has an empty body).



**Types of Constructors**

* [**Default Constructors:**](https://www.geeksforgeeks.org/c-internals-default-constructors-set-1/) Default constructor is the constructor which doesn’t take any argument. It has no parameters

|  |
| --- |
| // Cpp program to illustrate the  // concept of Constructors  #include <iostream>  using namespace std;    class construct {  public:      int a, b;        // Default Constructor      construct()      {          a = 10;          b = 20;      }  };    int main()  {      // Default constructor called automatically      // when the object is created      construct c;      cout << "a: " << c.a << endl           << "b: " << c.b;      return 1;  } |

* Output:
* a: 10
* b: 20
* **Note:**Even if we do not define any constructor explicitly, the compiler will automatically provide a default constructor implicitly.
* **Parameterized Constructors:**It is possible to pass arguments to constructors. Typically, these arguments help initialize an object when it is created. To create a parameterized constructor, simply add parameters to it the way you would to any other function. When you define the constructor’s body, use the parameters to initialize the object.

**Example=**

|  |
| --- |
| **// CPP program to illustrate**  **// parameterized constructors**  **#include <iostream>**  **using namespace std;**    **class Point {**  **private:**  **int x, y;**    **public:**  **// Parameterized Constructor**  **Point(int x1, int y1)**  **{**  **x = x1;**  **y = y1;**  **}**    **int getX()**  **{**  **return x;**  **}**  **int getY()**  **{**  **return y;**  **}**  **};**    **int main()**  **{**  **// Constructor called**  **Point p1(10, 15);**    **// Access values assigned by constructor**  **cout << "p1.x = " << p1.getX() << ", p1.y = " << p1.getY();**    **return 0;**  **}** |

Output:

p1.x = 10, p1.y = 15

When an object is declared in a parameterized constructor, the initial values have to be passed as arguments to the constructor function. The normal way of object declaration may not work. The constructors can be called explicitly or implicitly.

Example e = Example(0, 50); // Explicit call

Example e(0, 50); // Implicit call

**Uses of Parameterized constructor:**

* It is used to initialize the various data elements of different objects with different values when they are created.
* It is used to overload constructors.
* **Copy Constructor:** A copy constructor is a member function which initializes an object using another object of the same class. Detailed article on [Copy Constructor](https://www.geeksforgeeks.org/copy-constructor-in-cpp/).

Whenever we define one or more non-default constructors( with parameters ) for a class, a default constructor( without parameters ) should also be explicitly defined as the compiler will not provide a default constructor in this case. However, it is not necessary but it’s considered to be the best practice to always define a default constructor.

Copy constructor prototype-

**ClassName (const ClassName &old\_obj);**

**Copy constructor while calling:**

**First way-**

**classname new\_obj(old obj);**

**second ways-**

**classname new obj=old obj;**

|  |
| --- |
|  |

**Constructor overloading**

As we know function overloading is one of the core feature of the object oriented languages. We can use the same name of the functions; whose parameter sets are different. Here we will see how to overload the constructors of C++ classes. The constructor overloading has few important concepts.

* **Overloaded constructors must have the same name and different number of arguments**
* **The constructor is called based on the number and types of the arguments are passed.**
* **We have to pass the argument while creating objects, otherwise the constructor cannot understand which constructor will be called.**

**Example-#include <iostream>**

**using namespace std;**

**class Rect{**

**private:**

**int area;**

**public:**

**Rect(){**

**area = 0;**

**}**

**Rect(int a, int b){**

**area = a \* b;**

**}**

**void display(){**

**cout << "The area is: " << area << endl;**

**}**

**};**

**main(){**

**Rect r1;**

**Rect r2(2, 6);**

**r1.display();**

**r2.display();**

**}**

**----------------------------------------------------------------------------------------**

**Destructor in c++**

A destructor is a special member function that works just opposite to constructor, unlike [constructors](https://beginnersbook.com/2017/08/cpp-constructors/) that are used for initializing an object, destructors destroy (or delete) the object.

**Syntax of Destructor**

~class\_name()

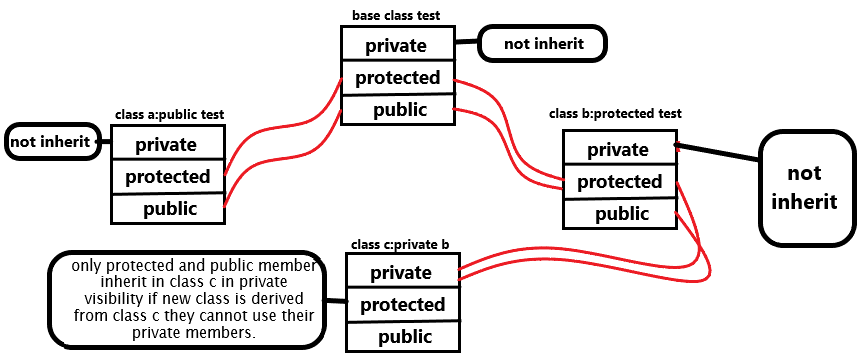
{

//Some code

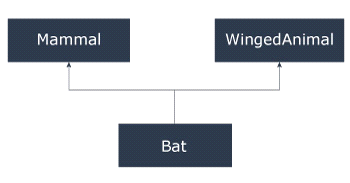
}

A destructor is **automatically called** when:  
**1) The program finished execution.  
2) When a scope (the { } parenthesis) containing**[**local variable**](https://beginnersbook.com/2017/08/cpp-variables/)**ends.  
3) When you call the delete operator.**

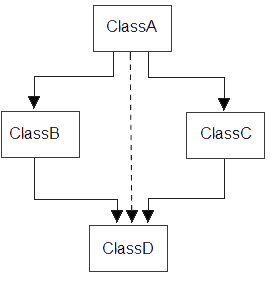
**Destructor rules**

* Name should begin with tilde sign(~) and must match class name.  
  2) There cannot be more than one destructor in a class.  
  3) Unlike constructors that can have parameters, destructors do not allow any parameter.  
  4) They do not have any return type, just like constructors.  
  5) When you do not specify any destructor in a class, compiler generates a default destructor and inserts it into your code.
* Destructor in c++
* A destructor is a special member function that works just opposite to constructor, unlike [constructors](https://beginnersbook.com/2017/08/cpp-constructors/) that are used for initializing an object, destructors destroy (or delete) the object.
* Syntax of Destructor
* ~class\_name()
* {
* //Some code
* }
* A destructor is **automatically called** when:  
  1) The program finished execution.  
  2) When a scope (the { } parenthesis) containing [**local variable**](https://beginnersbook.com/2017/08/cpp-variables/) ends.  
  3) When you call the delete operator.
* Destructor rules
* Name should begin with tilde sign(~) and must match class name.  
  2) There cannot be more than one destructor in a class.  
  3) Unlike constructors that can have parameters, destructors do not allow any parameter.  
  4) They do not have any return type, just like constructors.  
  5) When you do not specify any destructor in a class, compiler generates a default destructor and inserts it into your code.
* \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_
* **Visibility modes in c++**
* A derived class can access all the non-private members of its base class. Thus base-class members that should not be accessible to the member functions of derived classes should be declared private in the base class.
* 
* \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

**C++ Inheritance**

* One of the most important concepts in object-oriented programming is that of inheritance. Inheritance allows us to define a class in terms of another class, which makes it easier to create and maintain an application. This also provides an opportunity to reuse the code functionality and fast implementation time.
* When creating a class, instead of writing completely new data members and member functions, the programmer can designate that the new class should inherit the members of an existing class. This existing class is called the **base** class, and the new class is referred to as the **derived** class.
* The idea of inheritance implements the **is a** relationship. For example, mammal IS-A animal, dog IS-A mammal hence dog IS-A animal as well and so on.
* Base and Derived Classes
* A class can be derived from more than one classes, which means it can inherit data and functions from multiple base classes. To define a derived class, we use a class derivation list to specify the base class(es). A class derivation list names one or more base classes and has the form –
* Syntax-
* **class derived-class: access-specifier base-class**
* Where access-specifier is one of **public, protected,** or **private**, and base-class is the name of a previously defined class. If the access-specifier is not used, then it is private by default.
* **Implementation of Inheritance in C++ Programming**
* **C++ Single inheritance**
* class Person
* {
* ... .. ...
* };
* class MathsTeacher : public Person
* {
* ... .. ...
* };
* class Footballer : public Person
* {
* .... .. ...
* };
* There are various models of inheritance in C++ programming.
* **C++ Multilevel Inheritance**
* In C++ programming, not only you can derive a class from the base class but you can also derive a class from the derived class. This form of inheritance is known as multilevel inheritance.
* class A
* {
* ... .. ...
* };
* class B: public A
* {
* ... .. ...
* };
* class C: public B
* {
* ... ... ...
* };
* **C++ Multiple Inheritance**
* In C++ programming, a class can be derived from more than one parents. For example: A class *Bat* is derived from base classes *Mammal* and *WingedAnimal*. It makes sense because bat is a mammal as well as a winged animal.
* 
* **C++ Hierarchical Inheritance**
* If more than one class is inherited from the base class, it's known as [hierarchical inheritance](http://www.programtopia.net/cplusplus/docs/hierarchical-inheritance-c-programming?utm_source=programiz&utm_campaign=display). In hierarchical inheritance, all features that are common in child classes are included in the base class.
* For example: Physics, Chemistry, Biology are derived from Science class
* **Syntax of Hierarchical Inheritance**
* class base\_class {
* ... .. ...
* }
* class first\_derived\_class: public base\_class {
* ... .. ...
* }
* class second\_derived\_class: public base\_class {
* ... .. ...
* }
* class third\_derived\_class: public base\_class {
* ... .. ...
* }

Virtual base class inheritance

**A special case of hybrid inheritance : Multipath inheritance**:  
A derived class with two base classes and these two base classes have one common base class is called multipath inheritance. An ambiguity can arrise in this type of inheritance.  
  
Consider the following program:

|  |
| --- |
| **// C++ program demonstrating ambiguity in Multipath Inheritance**    **#include<iostream.h>**  **#include<conio.h>**  **class ClassA**  **{**  **public:**  **int a;**  **};**    **class ClassB : public ClassA**  **{**  **public:**  **int b;**  **};**  **class ClassC : public ClassA**  **{**  **public:**  **int c;**  **};**    **class ClassD : public ClassB, public ClassC**  **{**  **public:**  **int d;**  **};**    **void main()**  **{**    **ClassD obj;**    **//obj.a = 10;                   //Statement 1, Error**  **//obj.a = 100;                 //Statement 2, Error**    **obj.ClassB::a = 10;        //Statement 3**  **obj.ClassC::a = 100;      //Statement 4**    **obj.b = 20;**  **obj.c = 30;**  **obj.d = 40;**    **cout<< "\n A from ClassB  : "<< obj.ClassB::a;**  **cout<< "\n A from ClassC  : "<< obj.ClassC::a;**    **cout<< "\n B : "<< obj.b;**  **cout<< "\n C : "<< obj.c;**  **cout<< "\n D : "<< obj.d;**    **}** |

Output:

A from ClassB : 10

A from ClassC : 100

B : 20

C : 30

D : 40

In the above example, both ClassB & ClassC inherit ClassA, they both have single copy of ClassA. However ClassD inherit both ClassB & ClassC, therefore ClassD have two copies of ClassA, one from ClassB and another from ClassC.  
If we need to access the data member a of ClassA through the object of ClassD, we must specify the path from which a will be accessed, whether it is from ClassB or ClassC, bco’z compiler can’t differentiate between two copies of ClassA in ClassD.

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_  
  
**There are 2 ways to avoid this ambiguity:**

* **Use scope resolution operator**
* **Use virtual base class**

**Avoiding ambiguity using scope resolution operator:**  
Using scope resolution operator we can manually specify the path from which data member a will be accessed, as shown in statement 3 and 4, in the above example.

|  |
| --- |
| obj.ClassB::a = 10;        //Statement 3  obj.ClassC::a = 100;      //Statement 4 |

Note : Still, there are two copies of ClassA in ClassD.  
  
**Avoiding ambiguity using virtual base class:**

|  |
| --- |
| **#include<iostream.h>**  **#include<conio.h>**    **class ClassA**  **{**  **public:**  **int a;**  **};**    **class ClassB : virtual public ClassA**  **{**  **public:**  **int b;**  **};**  **class ClassC : virtual public ClassA**  **{**  **public:**  **int c;**  **};**    **class ClassD : public ClassB, public ClassC**  **{**  **public:**  **int d;**  **};**    **void main()**  **{**    **ClassD obj;**    **obj.a = 10;        //Statement 3**  **obj.a = 100;      //Statement 4**    **obj.b = 20;**  **obj.c = 30;**  **obj.d = 40;**    **cout<< "\n A : "<< obj.a;**  **cout<< "\n B : "<< obj.b;**  **cout<< "\n C : "<< obj.c;**  **cout<< "\n D : "<< obj.d;**    **}** |

Output:

A : 100

B : 20

C : 30

D : 40

According to the above example, ClassD has only one copy of ClassA, therefore, statement 4 will overwrite the value of a, given at statement 3.

**Pointers in C++**

Pointer is a variable in C++ that holds the address of another variable. They have [data type](https://beginnersbook.com/2017/08/cpp-data-types/) just like variables, for example an integer type pointer can hold the address of an integer variable and an character type pointer can hold the address of char variable.

***Syntax of pointer***

data\_type \*pointer\_name;

**How to declare a pointer?**

/\* This pointer p can hold the address of an integer

\* variable, here p is a pointer and var is just a

\* simple integer variable

\*/

int \*p, var

**Assignment**  
As I mentioned above, an integer type pointer can hold the address of another int variable. Here we have an integer variable var and pointer p holds the address of var. To assign the address of variable to pointer we use **ampersand symbol** (&).

/\* This is how you assign the address of another variable

\* to the pointer

\*/

p = &var;

**How to use it?**

// This will print the address of variable var

cout<<&var;

/\* This will also print the address of variable

\* var because the pointer p holds the address of var

\*/

cout<<p;

/\* This will print the value of var, This is

\* important, this is how we access the value of

\* variable through pointer

\*/

cout<<\*p;

**Example of Pointer**

Lets take a simple example to understand what we discussed above.

#include <iostream>

using namespace std;

int main(){

//Pointer declaration

int \*p, var=101;

//Assignment

p = &var;

cout<<"Address of var: "<<&var<<endl;

cout<<"Address of var: "<<p<<endl;

cout<<"Address of p: "<<&p<<endl;

cout<<"Value of var: "<<\*p;

return 0;

}

**Output:**

Address of var: 0x7fff5dfffc0c

Address of var: 0x7fff5dfffc0c

Address of p: 0x7fff5dfffc10

Value of var: 101

C++ Pointer to Pointer (Multiple Indirection)

A pointer to a pointer is a form of multiple indirection or a chain of pointers. Normally, a pointer contains the address of a variable. When we define a pointer to a pointer, the first pointer contains the address of the second pointer, which points to the location that contains the actual value as shown below.



A variable that is a pointer to a pointer must be declared as such. This is done by placing an additional asterisk in front of its name. For example, following is the declaration to declare a pointer to a pointer of type int −

int \*\*var;

**Pointer and arrays**

While handling [arrays](https://beginnersbook.com/2017/08/cpp-arrays/) with pointers you need to take care few things. First and very important point to note regarding arrays is that the array name alone represents the base address of array so while assigning the address of array to pointer don’t use ampersand sign(&). Do it like this:  
**Correct:** Because arr represent the address of array.

p = arr;

**Incorrect:**

p = &arr;

**Example: Traversing the array using Pointers**

#include <iostream>

using namespace std;

int main(){

//Pointer declaration

int \*p;

//Array declaration

int arr[]={1, 2, 3, 4, 5, 6};

//Assignment

p = arr;

for(int i=0; i<6;i++){

    cout<<\*p<<endl;

//++ moves the pointer to next int position

    p++;

   }

return 0;

}

**Output:**

1

2

3

4

5

6

**Pointer to Object**

In C++ you can declare a pointer that contains the address of the object of type class.

#include <iostream.h>  
class Base  
{  
public:  
int x;  
void display ()  
{  
cout<<"X="<<x<<endl;  
}  
};  
int main ()  
{  
Base B1;  
Base \*ptr;  
ptr = &B1;  
ptr->x = 10;  
ptr->display();  
}  
**Output:**  
X= 10

**Operator overloading**

You can redefine or overload most of the built-in operators available in C++. Thus, a programmer can use operators with user-defined types as well.

Overloaded operators are functions with special names: the keyword "operator" followed by the symbol for the operator being defined. Like any other function, an overloaded operator has a return type and a parameter list.

**Defining operator overloading outside the class-**

**Return type classname::operator symbol(argument list)**

**{**

**Body----------**

**}**

There is no argument needed to make unary operator overloading-

Test operator +()

There is only one argument needed to make binary operator overloading-

Test operator-(test)

There is one argument required to make friend unary operator overloading-

Friend test operator +(test)

There is two argument required to make friend binary operator overloading-

Friend test operator-(test,test)

**Calling operator overloading**

**Unary operator-**

Objname opsymbol;

opsymbol objname;

**Binary operator**

Objname1 opsymbol objname2

**Overloading Unary Operator**: Let us consider to overload (-) unary operator. In unary operator function, no arguments should be passed. It works only with one class objects. It is a overloading of an operator operating on a single operand.

**Overloading Binary Operator**: In binary operator overloading function, there should be one argument to be passed. It is overloading of an operator operating on two operands.

**Constructor in derived class**

[Constructors](https://www.geeksforgeeks.org/constructors-c/)  
Whenever we create an object of a class, the default constructor of that class is invoked automatically to initialize the members of the class.  
If we inherit a class from another class and create an object of the derived class, it is clear that the default constructor of the derived class will be invoked but before that the default constructor of all of the base classes will be invoke, i.e the order of invokation is that the base class’s default constructor will be invoked first and then the derived class’s default constructor will be invoked.

**Order of constructor call for Multiple Inheritance**

For multiple inheritance order of constructor call is, the base class’s constructors are called in the order of inheritance and then the derived class’s constructor.

filter\_none

edit

play\_arrow

brightness\_4

|  |
| --- |
| // C++ program to show the order of constructor calls  // in Multiple Inheritance    #include <iostream>  using namespace std;    // first base class  class Parent1  {        public:        // first base class's Constructor      Parent1()      {          cout << "Inside first base class" << endl;      }  };    // second base class  class Parent2  {      public:        // second base class's Constructor      Parent2()      {          cout << "Inside second base class" << endl;      }  };    // child class inherits Parent1 and Parent2  class Child : public Parent1, public Parent2  {      public:        // child class's Constructor      Child()      {          cout << "Inside child class" << endl;      }  };    // main function  int main() {        // creating object of class Child      Child obj1;      return 0;  } |

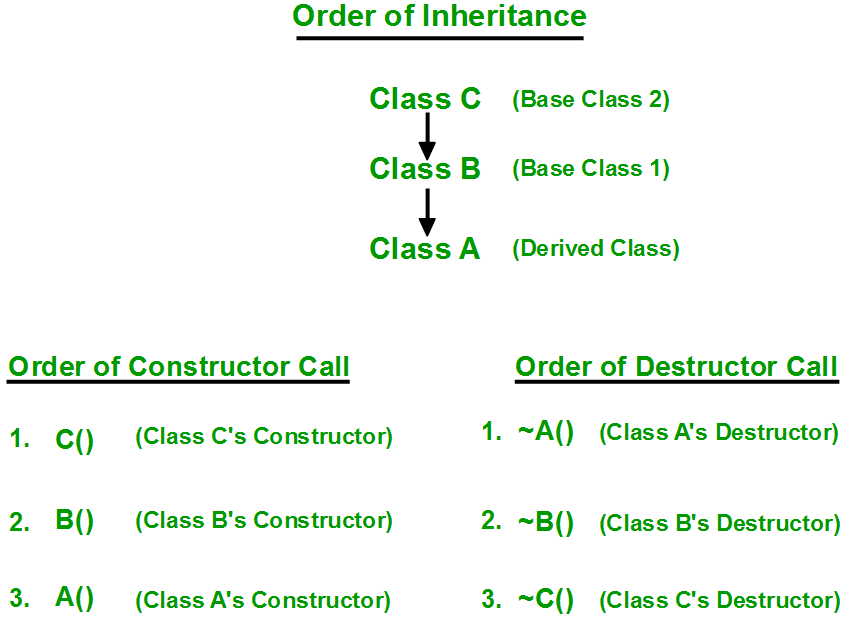
Output:

Inside first base class

Inside second base class

Inside child class

**Order of constructor and Destructor call for a given order of Inheritance**



**How to call the parameterized constructor of base class in derived class constructor?**

To call the parameterised constructor of base class when derived class’s parameterised constructor is called, you have to explicitly specify the base class’s parameterised constructor in derived class as shown in below program:

|  |
| --- |
| // C++ program to show how to call parameterised Constructor  // of base class when derived class's Constructor is called  #include <iostream>  using namespace std;    // base class  class Parent  {        public:        // base class's parameterised constructor      Parent(int i)      {  int x =i;          cout << "Inside base class's parameterised constructor" << endl;      }  };      // sub class  class Child : public Parent  {      public:        // sub class's parameterised constructor      Child(int j): Parent(j)      {          cout << "Inside sub class's parameterised constructor" << endl;      }  };    // main function  int main() {        // creating object of class Child      Child obj1(10);      return 0;  } |

Output:

Inside base class's parameterised constructor

Inside sub class's parameterised constructor

**Important Points**:

* Whenever the derived class’s default constructor is called, the base class’s default constructor is called automatically.
* To call the parameterised constructor of base class inside the parameterised consructor of sub class, we have to mention it explicitly.
* The parameterised constructor of base class cannot be called in default constructor of sub class, it should be called in the parameterised constructor of sub class.

**Initialization list of constructor in c++**

**1) For initialization of non-static const data members:**  
const data members must be initialized using Initializer List. In the following example, “t” is a const data member of Test class and is initialized using Initializer List. Reason for initializing the const data member in initializer list is because no memory is allocated separately for const data member, it is folded in the symbol table due to which we need to initialize it in the initializer list.  
Also, it is a copy constructor and we don’t need to call the assignment operator which means we are avoiding one extra operation.

#include<iostream>

using namespace std;

class Test {

    const int t;

public:

    Test(int t):t(t) {}  //Initializer list must be used

    int getT() { return t; }

};

int main() {

    Test t1(10);

    cout<<t1.getT();

    return 0;

}

* **/\* OUTPUT:**
* **10**
* **\*/**

this’ pointer in C++

To understand ‘this’ pointer, it is important to know how objects look at functions and data members of a class.

* Each object gets its own copy of the data member.
* All-access the same function definition as present in the code segment.

Meaning each object gets its own copy of data members and all objects share a single copy of member functions.  
Then now question is that if only one copy of each member function exists and is used by multiple objects, how are the proper data members are accessed and updated?  
The compiler supplies an implicit pointer along with the names of the functions as ‘this’.  
The ‘this’ pointer is passed as a hidden argument to all nonstatic member function calls and is available as a local variable within the body of all nonstatic functions. ‘this’ pointer is not available in static member functions as static member functions can be called without any object (with class name).  
For a class X, the type of this pointer is ‘X\* ‘. Also, if a member function of X is declared as const, then the type of this pointer is ‘const X \*’ (see [this HYPERLINK "https://www.geeksforgeeks.org/g-fact-77/" HYPERLINK "https://www.geeksforgeeks.org/g-fact-77/" HYPERLINK "https://www.geeksforgeeks.org/g-fact-77/"GFact](https://www.geeksforgeeks.org/g-fact-77/))

In the early version of C++ would let ‘this’ pointer to be changed; by doing so a programmer could change which object a method was working on. This feature was eventually removed, and now this in C++ is an r-value.  
**C++ lets object destroy themselves by calling the following code :**

|  |
| --- |
| **delete this;** |

As Stroustrup said ‘this’ could be the reference than the pointer, but the reference was not present in the early version of C++. If ‘this’ is implemented as a reference then, the above problem could be avoided and it could be safer than the pointer.

Following are the situations where ‘this’ pointer is used:

**1) When local variable’s name is same as member’s name**

|  |
| --- |
| #include<iostream>  using namespace std;  /\* local variable is same as a member's name \*/  class Test  {  private:     int x;  public:     void setX (int x)     {         // The 'this' pointer is used to retrieve the object's x         // hidden by the local variable 'x'         this->x = x;     }     void print() { cout << "x = " << x << endl; }  };  int main()  {     Test obj;     int x = 20;     obj.setX(x);     obj.print();     return 0;  } |

Output:

x = 20

For constructors, initializer list can also be used when parameter name is same as member’s name.

**2) To return reference to the calling object**

|  |
| --- |
| **/\* Reference to the calling object can be returned \*/**  **Test& Test::func ()**  **{**  **// Some processing**  **return \*this;**  **}** |

When a reference to a local object is returned, the returned reference can be used to **chain function calls** on a single object.

|  |
| --- |
| #include<iostream>  using namespace std;    class Test  {  private:    int x;    int y;  public:    Test(int x = 0, int y = 0) { this->x = x; this->y = y; }    Test &setX(int a) { x = a; return \*this; }    Test &setY(int b) { y = b; return \*this; }    void print() { cout << "x = " << x << " y = " << y << endl; }  };    int main()  {    Test obj1(5, 5);      // Chained function calls.  All calls modify the same object    // as the same object is returned by reference    obj1.setX(10).setY(20);      obj1.print();    return 0;  } |

Output:

x = 10 y = 20

**\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_**

**Pointer to derived class**

**if we create a class and derive second class and then we create base class pointer then base pointer can only access base class derived member but not derived class member but we can point base pointer to derived class object.**

 to point derived class object with base class pointer we can use casting method -**((derived\*)base pointer)**

we can also create derived class pointer and then point derived class object.

in the below example we study how to point derived class as well as base class with base class pointer.

**#include<iostream>**

**using namespace std;**

**class bc**

**{**

**protected:**

**int b;**

**public:**

**void setb(int x)**

**{**

**b=x;**

**}**

**void disp(void)**

**{**

**cout<<"\nvalue of b"<<b;**

**}**

**};**

**class dc:public bc**

**{**

**int d;**

**public:**

**void setd(int y)**

**{**

**d=y;**

**}**

**void disp(void)**

**{**

**cout<<"\nvalue of d"<<d;**

**}**

**};**

**int main()**

**{**

**bc \*bp;**

**bc obj1;**

**bp=&obj1;**

**bp->setb(70);**

**cout<<"\nbase pointer points to base data member\n";**

**bp->disp();**

**dc obj2;**

**bp=&obj2;**

**bp->setb(400);**

**cout<<"\nnew value set of b";**

**bp->disp();**

**cout<<"\nnew pointer is create by derived class\n";**

**dc \*dp;**

**dp=&obj2;**

**dp->setd(89);**

**cout<<"\nderived class value of d";**

**dp->disp();**

**//using casting we can call derived class by base pointer**

**cout<<"\nby using casting (dc\*)bp";**

**((dc\*)bp)->setd(47);**

**((dc\*)bp)->disp();**

**}**

**new and delete operators in C++ for dynamic memory**

**Dynamic memory allocation in C/C++ refers to performing memory allocation manually by programmer. Dynamically allocated memory is allocated on Heap and non-static and local variables get memory allocated on Stack (Refer Memory Layout C Programs for details).**

**What are applications?**

**One use of dynamically allocated memory is to allocate memory of variable size which is not possible with compiler allocated memory except variable length arrays.**

**The most important use is flexibility provided to programmers. We are free to allocate and deallocate memory whenever we need and whenever we don’t need anymore. There are many cases where this flexibility helps. Examples of such cases are Linked List, Tree, etc.**

**How is it different from memory allocated to normal variables?**

**For normal variables like “int a”, “char str[10]”, etc, memory is automatically allocated and deallocated. For dynamically allocated memory like “int \*p = new int[10]”, it is programmers responsibility to deallocate memory when no longer needed. If programmer doesn’t deallocate memory, it causes memory leak (memory is not deallocated until program terminates).**

**How is memory allocated/deallocated in C++?**

**C uses malloc() and calloc() function to allocate memory dynamically at run time and uses free() function to free dynamically allocated memory. C++ supports these functions and also has two operators new and delete that perform the task of allocating and freeing the memory in a better and easier way.**

**new operator**

**The new operator denotes a request for memory allocation on the Free Store. If sufficient memory is available, new operator initializes the memory and returns the address of the newly allocated and initialized memory to the pointer variable.**

**Syntax to use new operator: To allocate memory of any data type, the syntax is:**

**pointer-variable = new data-type;**

**Here, pointer-variable is the pointer of type data-type. Data-type could be any built-in data type including array or any user defined data types including structure and class.**

**Example:**

**// Pointer initialized with NULL**

**// Then request memory for the variable**

**int \*p = NULL;**

**p = new int;**

**OR**

**// Combine declaration of pointer**

**// and their assignment**

**int \*p = new int;**

**Initialize memory**: We can also initialize the memory using new operator:

pointer-variable = new data-type(value);

**Example:**

**int \*p = new int(25);**

**float \*q = new float(75.25);**

Allocate block of memory: new operator is also used to allocate a block(an array) of memory of type data-type.

**pointer-variable = new data-type[size];**

**where size(a variable) specifies the number of elements in an array.**

**Example:**

**int \*p = new int[10]**

Dynamically allocates memory for 10 integers continuously of type int and returns pointer to the first element of the sequence, which is assigned to p(a pointer). p[0] refers to first element, p[1] refers to second element and so on.

**delete operator**

Since it is programmer’s responsibility to deallocate dynamically allocated memory, programmers are provided delete operator by C++ language.

Syntax:

// Release memory pointed by pointer-variable

delete pointer-variable;

Here, pointer-variable is the pointer that points to the data object created by new.

Examples:

delete p;

delete q;

To free the dynamically allocated array pointed by pointer-variable, use following form of delete:

// Release block of memory

// pointed by pointer-variable

delete[ ] pointer-variable;

Example:

// It will free the entire array

// pointed by p.

delete[ ] p;

**example**

#include<iostream>

using namespace std;

int main()

{

int n,i;

float \*p,total=0;

cout<<"\nenter number of subject-";

cin>>n;

p=new float[n];

cout<<"enter subject marks";

for(i=0;i<n;i++)

{

cout<<"\nsubject "<<(i+1)<<"\n";

cin>>p[i];

cout<<"\nmarks of";

}

for(i=0;i<n;i++)

{

cout<<"subject"<<(i+1)<<"\n";

cout<<\*(p+i);

total=total+\*(p+i);

}

cout<<"\n total-"<<total;

}

Virtual Function in C++

A virtual function is a member function which is declared within a base class and is re-defined(Overriden) by a derived class. When you refer to a derived class object using a pointer or a reference to the base class, you can call a virtual function for that object and execute the derived class’s version of the function.

* Virtual functions ensure that the correct function is called for an object, regardless of the type of reference (or pointer) used for function call.
* They are mainly used to achieve Runtime polymorphism
* Functions are declared with a virtual keyword in base class.
* The resolving of function call is done at Run-time.

**Rules for Virtual Functions**

1. Virtual functions cannot be static and also cannot be a friend function of another class.
2. Virtual functions should be accessed using pointer or reference of base class type to achieve run time polymorphism.
3. The prototype of virtual functions should be same in base as well as derived class.
4. They are always defined in base class and overridden in derived class. It is not mandatory for derived class to override (or re-define the virtual function), in that case base class version of function is used
5. A class may have virtual destructor but it cannot have a virtual constructor.

#include<iostream>

using namespace std;

class base

{

public:

void disp(void)

{

cout<<"\nbase class disp function";

}

virtual void show(void)

{

cout<<"\nbase class virtual show function";

}

};

class derived:public base

{

void disp(void)

{

cout<<"\nderived class disp function";

}

virtual void show(void)

{

cout<<"\nderived class virtual show function";

}

};

int main()

{

base obj;

derived obj2;

base \*b;

b=&obj;

b->disp();

b->show();

b=&obj2;

b->disp();

b->show();

}

**\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_**

**Pure Virtual Functions and Abstract Classes in C++**

Sometimes implementation of all function cannot be provided in a base class because we don’t know the implementation. Such a class is called abstract class. For example, let Shape be a base class. We cannot provide implementation of function draw() in Shape, but we know every derived class must have implementation of draw(). Similarly an Animal class doesn’t have implementation of move() (assuming that all animals move), but all animals must know how to move. We cannot create objects of abstract classes.

A pure virtual function (or abstract function) in C++ is a virtual function for which we don’t have implementation, we only declare it. A pure virtual function is declared by assigning 0 in declaration.

**rules-**

* if pure virtual function in base class then it is called abstract class.
* no object is create in abstract class but in c++ we can create pointer of abstract class whose hold the address of derived class.
* if any derived class is inherited by abstract class then it is responsiblity of derived class to give defination of base pure virtual function but if derived class not give pure virtual class defination it is called derived abstract class.

example-

#include<iostream>

using namespace std;

class base

{

public:

virtual void disp(void)=0;

};

class derived:public base

{

public:

void disp()

{

cout<<"\nderived classes";

}

};

int main()

{

//in abstract class object are not created but can create pointer of abstract class

//if any class contain pure virtual function then it is called abstract class

base \*ptr;

derived obj2;

obj2.disp();

ptr=&obj2;

ptr->disp();

}

***Some Interesting Facts:***

**1) A class is abstract if it has at least one pure virtual function.**

**In the following example, Test is an abstract class because it has a pure virtual function show().**

// pure virtual functions make a class abstract

**#include<iostream>**

**using namespace std;**

**class Test**

**{**

**int x;**

**public:**

**virtual void show() = 0;**

**int getX() { return x; }**

**};**

**int main(void)**

**{**

**Test t;**

**return 0;**

**}**

Output:

Compiler Error: cannot declare variable 't' to be of abstract

type 'Test' because the following virtual functions are pure

within 'Test': note: virtual void Test::show()

**2) We can have pointers and references of abstract class type.**

**For example the following program works fine.**

#include<iostream>

using namespace std;

class Base

{

public:

virtual void show() = 0;

};

class Derived: public Base

{

public:

void show() { cout << "In Derived \n"; }

};

int main(void)

{

Base \*bp = new Derived();

bp->show();

return 0;

}

Output:

In Derived

**3) If we do not override the pure virtual function in derived class, then derived class also becomes abstract class.**

**The following example demonstrates the same.**

#include<iostream>

using namespace std;

class Base

{

public:

virtual void show() = 0;

};

class Derived : public Base { };

int main(void)

{

Derived d;

return 0;

}

Compiler Error: cannot declare variable 'd' to be of abstract type

'Derived' because the following virtual functions are pure within

'Derived': virtual void Base::show()

**4) An abstract class can have constructors.**

**For example, the following program compiles and runs fine.**

#include<iostream>

using namespace std;

// An abstract class with constructor

class Base

{

protected:

int x;

public:

virtual void fun() = 0;

Base(int i) { x = i; }

};

class Derived: public Base

{

int y;

public:

Derived(int i, int j):Base(i) { y = j; }

void fun() { cout << "x = " << x << ", y = " << y; }

};

int main(void)

{

Derived d(4, 5);

d.fun();

return 0;

}

Output:

x = 4, y = 5

**polymrphism in c++**

**The word polymorphism means having many forms. In simple words, we can define polymorphism as the ability of a message to be displayed in more than one form**.

Real life example of polymorphism, a person at the same time can have different characteristic. Like a man at the same time is a father, a husband, an employee. So the same person posses different behavior in different situations. This is called polymorphism.

Polymorphism is considered as one of the important features of Object Oriented Programming.

In C++ polymorphism is mainly divided into two types:

1. Compile time Polymorphism
2. Runtime Polymorphism

**Compile time polymorphism**: This type of polymorphism is achieved by function overloading or operator overloading.

**Function Overloading:** When there are multiple functions with same name but different parameters then these functions are said to be overloaded. Functions can be overloaded by change in number of arguments or/and change in type of arguments.

**Operator Overloading:** C++ also provide option to overload operators. For example, we can make the operator (‘+’) for string class to concatenate two strings. We know that this is the addition operator whose task is to add two operands. So a single operator ‘+’ when placed between integer operands , adds them and when placed between string operands, concatenates them.

**Runtime polymorphism**: This type of polymorphism is achieved by Function Overriding and pure virtual function.

Function overriding on the other hand occurs when a derived class has a definition for one of the member functions of the base class. That base function is said to be overridden.

* in below runtime polymorphism is achieved by pure virtual function.

**example-**

#include<iostream>

using namespace std;

class base

{

public:

virtual void disp()=0;

};

class derived:public base

{

int a;

public:

derived(int x)

{

a=x;

}

void disp(void)

{

cout<<"\na value in class derived"<<a;

}

};

class derived2:public base

{

int b;

public:

derived2(int y)

{

b=y;

}

void disp(void)

{

cout<<"\nb value in class derived 2"<<b;

}

};

int main()

{

base \*p;

base \*pr[2];

derived obj(100);

derived2 obj2(200);

//we can achieve runtime polymorpism by pure virtual function in which we can decide which pure function defination run at the time of program run

p=&obj;

cout<<"pointer of abstract class initialized by class derived\n";

p->disp();

p=&obj2;

cout<<"\npointer of abstract class initialized by class derived2\t\n";

p->disp();

pr[0]=&obj;

cout<<"\npointer array pr[0] of abstract class initialized by class derived\t\n";

pr[1]=&obj2;

cout<<"\npointer array pr[1] of abstract class initialized by class derived2\t\n";

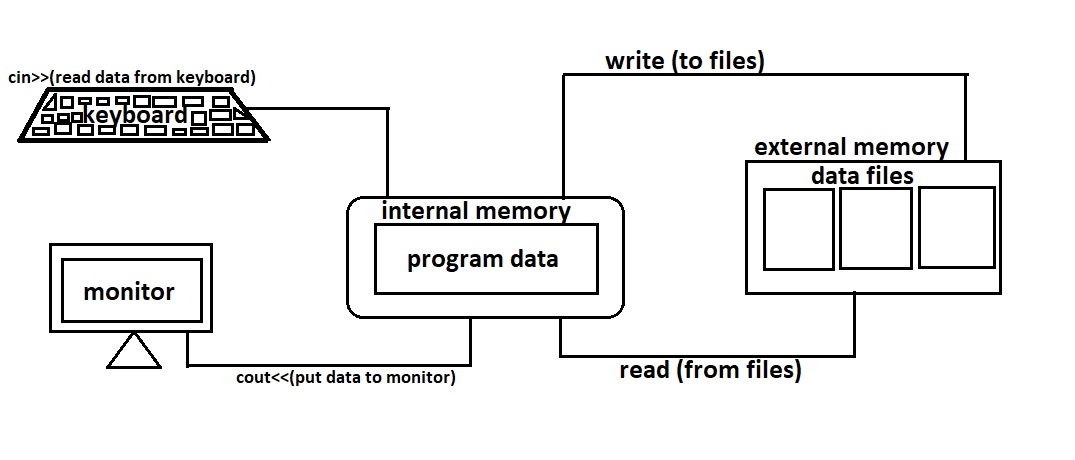
pr[0]->disp();

pr[1]->disp();

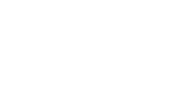
}

# File Handling through C++ Classes

In C++, files are mainly dealt by using three classes fstream, ifstream, ofstream available in fstream headerfile.  
 **ofstream:** Stream class to write on files  
 **ifstream:** Stream class to read from files  
 **fstream:** Stream class to both read and write from/to files.



|  |  |
| --- | --- |
| **Sr.No** | **Data Type & Description** |
| 1 | **ofstream**  This data type represents the output file stream and is used to create files and to write information to files. |
| 2 | **ifstream**  This data type represents the input file stream and is used to read information from files. |
| 3 | **fstream**  This data type represents the file stream generally, and has the capabilities of both ofstream and ifstream which means it can create files, write information to files, and read information from files. |



# **Standard Template Library(S.T.L)**



why we use STL?

* Arrays are limited due to **fixed size and manual management**.
* Real-world applications need **dynamic, efficient data handling**.
* **STL simplifies this** by offering advanced data structures (e.g., vectors, lists) with built-in resizing and algorithms.



**Components of STL**

The components of STL are the features provided by Standard Template Library (STL) in C++ that can be classified into 4 types:

1. **Containers**
2. **Algorithms**
3. **Iterators**
4. **Functors**

These components are designed to be efficient, flexible, and reusable, making them an integral part of modern C++ programming.

**Containers**

[**Containers**](https://www.geeksforgeeks.org/cpp/containers-cpp-stl/) are the data structures used to store objects and data according to the requirement. Each container is implemented as a template class that also contains the methods to perform basic operations on it. Every STL container is defined inside its own header file.

Containers can be further classified into 4 types:

1. **Sequence Containers**
2. **Container Adaptors**
3. **Associative Containers**
4. **Unordered Associated Containers**

If you want to dive deep into STL and understand its full potential, our[**Complete C++ Course**](https://www.geeksforgeeks.org/courses/cpp-programming-basic-to-advanced?utm_campaign=221_cpp_standard_template_library_stl&utm_medium=gfgcontent_icp&utm_source=geeksforgeeks) offers a complete guide to mastering containers, iterators, and algorithms provided by STL.

**Sequence Containers**

Sequence containers store the data in the linear manner. They are also used to implement container adaptors.  
There are 5 sequence containers in C++ STL:

1. [**Arrays**](https://www.geeksforgeeks.org/cpp/stdarray-in-cpp/): The STL array is an implementation of a compile time non-resizable array. It contains various method for common array operations.
2. [**Vector**](https://www.geeksforgeeks.org/cpp/vector-in-cpp-stl/): An STL vector can be defined as the dynamic sized array which can be resized automatically when new elements are added or removed.
3. [**Deque**](https://www.geeksforgeeks.org/cpp/deque-cpp-stl/): Deque or Double-Ended Queue is sequence containers with the feature of expansion and contraction on both ends. It means we can add and remove the data to and from both ends.
4. [**Lists**](https://www.geeksforgeeks.org/cpp/list-cpp-stl/): List container stores data in non-contiguous memory unlike vectors and only provide sequential access to the stored data. It basically implements the doubly linked list.
5. [**Forward Lists**](https://www.geeksforgeeks.org/cpp/forward-list-c-set-1-introduction-important-functions/)**:** Forward lists also store the data in a sequential manner like lists, but with the difference that forward list stores the location of only the next elements in the sequence. It implements the singly linked list.

| **Container** | **Header** | **Syntax Example** |
| --- | --- | --- |
| vector | <vector> | vector<int> v; |
| list | <list> | list<string> names; |
| deque | <deque> | deque<float> dq; |
| array | <array> | array<int, 5> arr; |
| forward\_list | <forward\_list> | forward\_list<int> fl; |

**Container Adaptors**

The container adapters are the type of STL containers that adapt existing container classes to suit specific needs or requirements.  
There are 3 container adaptors in C++ STL:

1. [**Stack**](https://www.geeksforgeeks.org/cpp/stack-in-cpp-stl/)**:** STL Stack follows the Last In First Out (LIFO) principle of element insertion and deletion. Also, these operations are performed only at one end of the stack.
2. [**Queue**](https://www.geeksforgeeks.org/cpp/queue-cpp-stl/)**:**STL Queue follows the First In First Out (FIFO) principle, means the element are inserted first are removed first and the elements inserted last are removed at last. It uses deque container by default.
3. [**Priority Queue**](https://www.geeksforgeeks.org/cpp/priority-queue-in-cpp-stl/)**:** STL Priority Queue does not follow any of the FIFO or LIFO principle, but the deletion of elements is done on the basis of its priority. So, the element with the highest (by default) is always removed first. By default, it uses vector as underlying container.

| **Adapter** | **Header** | **Syntax Example** |
| --- | --- | --- |
| stack | <stack> | stack<int> st; |
| queue | <queue> | queue<string> q; |
| priority\_queue | <queue> | priority\_queue<int> pq; |

**Associative Containers**

Associative containers are the type of containers that store the elements in a sorted order based on keys rather than their insertion order.  
There are 4 associative containers in C++ STL:

1. [**Sets**](https://www.geeksforgeeks.org/cpp/set-in-cpp-stl/): STL Set is a type of associative container in which each element has to be unique because the value of the element identifies it. By default, the values are stored in ascending order.
2. [**Maps**](https://www.geeksforgeeks.org/cpp/map-associative-containers-the-c-standard-template-library-stl/): STL Maps are associative containers that store elements in the form of a key-value pair. The keys have to be unique and the container is sorted on the basis of the values of the keys.
3. [**Multisets**](https://www.geeksforgeeks.org/cpp/multiset-in-cpp-stl/): STL Multiset is similar to the set container except that it can store duplicate values.
4. [**Multimaps**](https://www.geeksforgeeks.org/cpp/multimap-associative-containers-the-c-standard-template-library-stl/): STL Multimap is similar to a map container but allows multiple mapped values to have same keys.

| **Container** | **Header** | **Syntax Example** |
| --- | --- | --- |
| set | <set> | set<int> s; |
| map | <map> | map<string, int> m; |
| multiset | <set> | multiset<int> ms; |
| multimap | <map> | multimap<int, string> mm; |

**Unordered Associative Containers**

Unordered associative containers store the data in no particular order, but they allow the fastest insertion, deletion and search operations among all the container types in STL.  
There are 4 unordered associative containers in C++ STL:

1. [**Unordered Set**](https://www.geeksforgeeks.org/cpp/unordered_set-in-cpp-stl/): STL Unordered Set stores the unique keys in the form of hash table. The order is randomized but insertion, deletion and search are fast.
2. [**Unordered Multiset**](https://www.geeksforgeeks.org/cpp/cpp-unordered_multiset/): STL Unordered Multiset works similarly to an unordered set but can store multiple copies of the same key.
3. [**Unordered Map**](https://www.geeksforgeeks.org/cpp/unordered_map-in-cpp-stl/): STL Unordered Map stores the key-value pair in a hash table, where key is hashed to find the storage place.
4. [**Unordered Multimap**](https://www.geeksforgeeks.org/cpp/unordered_multimap-and-its-application/): STL Unordered Multimap container is similar to unordered map, but it allows multiple values mapped to the same key.

| **Container** | **Header** | **Syntax Example** |
| --- | --- | --- |
| unordered\_set | <unordered\_set> | unordered\_set<int> us; |
| unordered\_map | <unordered\_map> | unordered\_map<string, int> um; |
| unordered\_multiset | <unordered\_set> | unordered\_multiset<int> ums; |
| unordered\_multimap | <unordered\_map> | unordered\_multimap<int, string> umm; |

**Algorithms**

[**STL algorithms**](https://www.geeksforgeeks.org/cpp/c-magicians-stl-algorithms/) offer a wide range of functions to perform common operations on data (mainly containers). These functions implement the most efficient version of the algorithm for tasks such as sorting, searching, modifying and manipulating data in containers, etc. All STL algorithms are defined inside the**<algorithm>** and **<numeric>** header file.  
There is no formal classification of STL algorithms, but we can group them into two types based on the type of operations they perform:

#include <algorithm>

sort(v.begin(), v.end()); // Sort vector

reverse(v.begin(), v.end()); // Reverse vector

count(v.begin(), v.end(), 5); // Count occurrences of 5

find(v.begin(), v.end(), 10); // Find element 10

**Manipulative Algorithms**

Manipulative algorithms perform operations that modifies the elements of the given container or rearrange their order.  
Some of the common manipulative algorithm includes:

* [**copy**](https://www.geeksforgeeks.org/cpp/different-methods-copy-c-stl-stdcopy-copy_n-copy_if-copy_backward/): Copies a specific number of elements from one range to another.
* [**fill**](https://www.geeksforgeeks.org/cpp/fill-in-cpp-stl/): Assigns a specified value to all elements in a range.
* [**transform**](https://www.geeksforgeeks.org/cpp/transform-c-stl-perform-operation-elements/): Applies a function to each element in a range and stores the result in another range.
* [**replace**](https://www.geeksforgeeks.org/cpp/stdreplace-stdreplace_if-c/): Replaces all occurrences of a specific value in a range with a new value.
* [**swap**](https://www.geeksforgeeks.org/cpp/swap-in-cpp/): Exchanges the values of two variables.
* [**reverse**](https://www.geeksforgeeks.org/cpp/stdreverse-in-c/): Reverses the order of elements in a range.
* [**rotate**](https://www.geeksforgeeks.org/cpp/rotate-in-cpp-stl/): Rotates the elements in a range such that a specific element becomes the first.
* [**remove**](https://www.geeksforgeeks.org/cpp/std-remove-algorithm-in-cpp/): Removes all elements with a specified value from a range but does not reduce the container size.
* [**unique**](https://www.geeksforgeeks.org/cpp/stdunique-in-cpp/): Removes consecutive duplicate elements from a range.

**Non-Manipulative Algorithms**

Non-manipulating algorithms are the type of algorithms provided by the Standard Template Library (STL) that operate on elements in a range without altering their values or the order of the elements.  
The below are the few examples of the STL's non-manipulative algorithms:

* [**max\_element**](https://www.geeksforgeeks.org/cpp/max_element-in-cpp/): Find the maximum element in the given range.
* [**min\_element**](https://www.geeksforgeeks.org/cpp/stdmin_element-in-cpp/): To find the minimum element in the given range.
* [**accumulate**](https://www.geeksforgeeks.org/cpp/accumulate-and-partial_sum-in-c-stl-numeric-header/): Finds the sum of the elements of the given range.
* [**count**](https://www.geeksforgeeks.org/cpp/std-count-cpp-stl/): Counts the occurrences of given element in the range.
* [**find**](https://www.geeksforgeeks.org/cpp/std-find-in-cpp/): Returns an iterator to the first occurrence of an element in the range.
* [**is\_permutation**](https://www.geeksforgeeks.org/cpp/stdis_permutation-c-stl/): Checks if one range is a permutation of another.
* [**is\_sorted**](https://www.geeksforgeeks.org/cpp/stdis_sorted-in-cpp/): Checks if the elements in a range are sorted in non-decreasing order.
* [**partial\_sum**](https://www.geeksforgeeks.org/cpp/accumulate-and-partial_sum-in-c-stl-numeric-header/): Computes the cumulative sum of elements in a range.

**Iterators**

[**Iterators**](https://www.geeksforgeeks.org/cpp/introduction-iterators-c/) are the pointer like objects that are used to point to the memory addresses of STL containers. They are one of the most important components that contributes the most in connecting the STL algorithms with the containers. Iterators are defined inside the **<iterator>**header file.

**🔹 Types**

* begin() – points to first element
* end() – points past the last element
* rbegin() / rend() – reverse iterators

**🔹 Syntax Example**

cpp

vector<int>::iterator it;

for (it = v.begin(); it != v.end(); ++it)

cout << \*it << " ";

You can also use **auto** for cleaner syntax:

cpp

for (auto x : v)

cout << x << " ";

In C++ STL, iterators are of 5 types:

1. [**Input Iterators**](https://www.geeksforgeeks.org/cpp/input-iterators-in-cpp/): Input Iterators can be used to read values from a sequence once and only move forward.
2. [**Output Iterators**](https://www.geeksforgeeks.org/cpp/output-iterators-cpp/): Output Iterators can be used to write values into a sequence once and only move forward.
3. [**Forward Iterators**](https://www.geeksforgeeks.org/cpp/forward-iterators-in-cpp/): Forward Iterators combine the features of both input and output iterators.
4. [**Bidirectional Iterators**](https://www.geeksforgeeks.org/cpp/bidirectional-iterators-in-cpp/): Bidirectional Iterators support all operations of forward iterators and additionally can move backward.
5. [**Random Access Iterators**](https://www.geeksforgeeks.org/cpp/random-access-iterators-in-cpp/): Random Access Iterators support all operations of bidirectional iterators and additionally provide efficient random access to elements.

**Functors**

[Functors](https://www.geeksforgeeks.org/cpp/functors-in-cpp/) are objects that can be treated as though they are a function. Functors are most commonly used along with STL algorithms. It overloads the **function-call operator** () and allows us to use an object like a function. There are many predefined functors in C++ STL that are defined inside the **<functional>** header file.

Functors can be classified into multiple types based on the type of operator they perform:

1. **Arithmetic Functors**
2. **Relational Functors**
3. **Logical Functors**
4. **Bitwise Functors**

## Syntax of a Functor

cpp

#include <iostream>

using namespace std;

class Add {

public:

int operator()(int a, int b) {

return a + b;

}

};

int main() {

Add add; // Create functor object

cout << add(5, 3); // Call like a function

return 0;

}

### 🧾 Output:

Code

8

**Real-World Use Case: Custom Comparator in sort**

cpp

#include <iostream>

#include <vector>

#include <algorithm>

using namespace std;

class Descending {

public:

bool operator()(int a, int b) {

return a > b;

}

};

int main() {

vector<int> v = {3, 1, 4, 2};

sort(v.begin(), v.end(), Descending());

for (int x : v)

cout << x << " ";

return 0;

}

**🧾 Output:**

Code

4 3 2 1