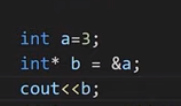
### Pointers in C++

A pointer is a data type which holds the address of other data type. The “**&**” operator is called “**address off**" operator, and the **"\*”** operator is called “**value at**” dereference operator.

“**&**” ----🡪 **address off** operator

**"\*”----🡪 value at** dereference operator

An example program for pointers is shown in figure 1.



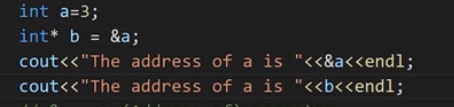
***Figure 1: Pointer Program***

As shown in figure 1, at 1st line an integer variable “**a**” is initialized with the value “**3**". At the 2nd line, the address of integer variable "**a**” is assigned to the integer pointer variable “**b**". At the 3rd line, the address of the integer pointer variable "**b**” is printed. The output of the following program is shown in figure 2.



***Figure 2: Pointer Program Output***

As shown in figure 2, the address of the integer pointer variable "**b**” is printed. The main thing to note here is that the address printed by the variable “**b**" is the address of integer variable "**a**” because we had assigned the address of variable “**a**” to the integer pointer variable “**b**". To clarify, we will print both variable "a" and variable "b" addresses, which are shown in figure 3.



***Figure 3: Pointer Program Example 2***

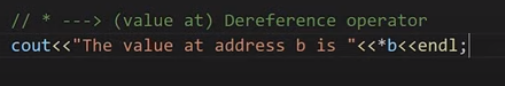
As shown in figure 3, now we printed both variable “**a**” and variable “**b**” addresses. The output for the following program is shown in figure 4.



***Figure 4: Pointer Program Example 2 Output***

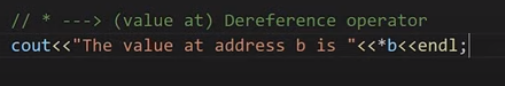
As shown in figure 4, both variables "**a**” and “**b**” have the same addresses, but in actual, this is the address of the variable "**a**”, the variable “**b**" is just pointing to the address of the variable "**a**”.

To see the value of variable “**a**" using a pointer variable, we can use the "**\***" dereference operator. An example of the dereference operator program is shown in figure 5.



***Figure 5: Dereference Operator example***

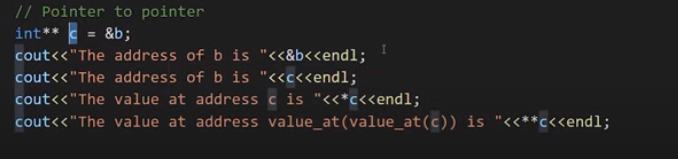
As shown in figure 5, the value at address “**b**” is printed. The main thing to note here is that the value printed by the pointer variable “**b**” will be the value of variable “**a**” because the pointer variable “**b**" is pointing to the address of the variable "**a**”. The output for the following program is shown in figure 6.



***Figure 6: Dereference Operator Example***

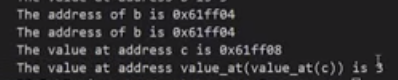
#### Pointer to Pointer

Pointer to Pointer is a simple concept, in which we store the address of one Pointer to another pointer. An example program for Pointer to Pointer is shown in figure 7.



***Figure 7: Pointer to Pointer Example Program***

As shown in figure 7, at the 1st line, the address of the pointer variable "**b**” is assigned to the pointer variable “**c**”. At 2nd line, the address of the pointer variable "**b**” is printed. At the 3rd line, the address of the pointer variable "**c**” is printed. At line 4th, the value at the pointer variable "**c**” is printed. At line 5th, the pointer variable "**c**” will be dereferenced two times, and it will print the value at pointer variable "**b**”. The output of the following program is shown in figure 2. The output for the following program is shown in figure 8.



***Figure 8: Pointer to Pointer Example Program Output***

**Arrays in C++**

In this tutorial, we will discuss arrays and pointer arithmetic in C++

**What are Arrays in C++:**

* An array is a collection of items which are of the similar type stored in contiguous memory locations.
* Sometimes, a simple variable is not enough to hold all the data.
* For example, let’s say we want to store the marks of 2500 students; initializing 2500 different variable for this task is not feasible.
* To solve this problem, we can define an array with size 2500 that can hold the marks of all students.
* For example **int marks[2500];**

**Why do we need arrays?**   
We can use normal variables (v1, v2, v3, ..) when we have a small number of objects, but if we want to store a large number of instances, it becomes difficult to manage them with normal variables. The idea of an array is to represent many instances in one variable.

**Advantages of an Array in C/C++:**

1. Random access of elements using array index.
2. Use of less line of code as it creates a single array of multiple elements.
3. Easy access to all the elements.
4. Traversal through the array becomes easy using a single loop.
5. Sorting becomes easy as it can be accomplished by writing less line of code.

**Disadvantages of an Array in C/C++:**

1. Allows a fixed number of elements to be entered which is decided at the time of declaration. Unlike a linked list, an array in C is not dynamic.
2. Insertion and deletion of elements can be costly since the elements are needed to be managed in accordance with the new memory allocation.

An example program for an array is shown in code snippet below.

int marks[] = {23, 45, 56, 89};

cout<<marks[0]<<endl;

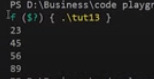
cout<<marks[1]<<endl;

cout<<marks[2]<<endl;

cout<<marks[3]<<endl;

***Code Snippet 1: Array Program 1***

As shown in the code snippet, we initialized an array of size 4 in which we have stored marks of 4 students and then printed them one by one. The main point to note here is that array store data in continuous block form in the memory, and array indexes start from 0. Output for the following program is shown in figure 1.



***Figure 1: Array Program 1 Output***

Another example program to declare an array is shown in code snippet 2.

int mathMarks[4];

mathMarks[0] = 2278;

mathMarks[1] = 738;

mathMarks[2] = 378;

mathMarks[3] = 578;

cout<<"These are math marks"<<endl;

cout<<mathMarks[0]<<endl;

cout<<mathMarks[1]<<endl;

cout<<mathMarks[2]<<endl;

cout<<mathMarks[3]<<endl;

***Code Snippet 2: Array Program 2***

As shown in code snippet 2, we have declared an array of size 4 and then assigned values one by one to each index of the array. Output for the following program is shown in figure 2.



***Figure 2: Array Program 2 Output***

To change the value at the specific index of an array, we can simply assign the value to that index. For example: “**marks[2] = 333**” can place the value “**333**” at the index “**2**” of the array. We can use loops to print the values of an array, instead of printing them one by one. An example program to print the value of the array with "for" loop is shown in code snippet 3.

for (int i = 0; i < 4; i++)

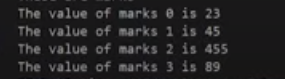
{

cout<<"The value of marks "<<i<<" is "<<marks[i]<<endl;

}

***Code Snippet 3: Array program with a loop***

As shown in code snippet 3, we initialized an integer variable “i" with the value 0 and set the running condition of the loop to the length of an array. In the loop body, each index number and the value at each number is being printed. Output for the following program is shown in figure 3.



***Figure 3: Array program with loop output***

**Q. Is declaration of Size of Array necessary ?**

- You don't have to specify the size of the array. But if you don't, it will only be as big as the elements that are inserted into it:

e.g: string cars[] = {"Volvo", "BMW", "Ford"}; // size of array is always 3

This is completely fine. However, the problem arise if you want extra space for future elements. Then you have to overwrite the existing values:

~~e.g; string cars[] = {"Volvo", "BMW", "Ford"};~~  
e.g: string cars[] = {"Volvo", "BMW", "Ford", "Mazda", "Tesla"};

If you specify the size however, the array will reserve the extra space:

e.g: string cars[5] = {"Volvo", "BMW", "Ford"}; // size of array is 5, even though it's only three elements inside it

Now you can add a fourth and fifth element without overwriting the others:

cars[3] = "Mazda";  
cars[4] = "Tesla";

**Pointers and Arrays:**

Storing the address of an array into pointer is different than storing the address of a variable into the pointer because the name of the array is an address of the first index of an array. So to use ampersand "&" with the array name for assigning the address to a pointer is wrong.

* &Marks --> Wrong
* Marks --> address of the first block

An example program for storing the starting address of an array in the pointer is shown in code snippet 4.

int\* p = marks;

cout<<"The value of marks[0] is "<<\*p<<endl;

***Code Snippet 4: Pointer and Array Program***

As shown in code snippet 7, we have assigned the address of array “marks” to the pointer variable “\*p” and then printed the pointer “\*p”. The main thing to note here is that the value at the pointer “\*p” is the starting address of the array “marks”. The output for the following program is shown in figure 4.



***Figure 4: Pointer and Array Program Output***

As shown in figure 4, we have printed the value at pointer "\*p", and it has shown us the value of the first index of the array "marks" because the pointer was pointing at the first index of an array and the value at that index was "23". If we want to access the 2nd index of an array through the pointer, we can simply increment the pointer with 1. For example: "**\*(p+1)**" will give us the value of the 2nd index of an array. An example program to print the values of an array through the pointer is shown in code snippet 5.

int\* p = marks;

cout<<"The value of \*p is "<<\*p<<endl;

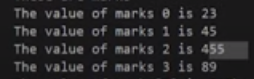
cout<<"The value of \*(p+1) is "<<\*(p+1)<<endl;

cout<<"The value of \*(p+2) is "<<\*(p+2)<<endl;

cout<<"The value of \*(p+3) is "<<\*(p+3)<<endl;

***Code Snippet 5: Pointer and Array Program 2***

As shown in code snippet 5, 1st we have printed the value at pointer “**\*p**”; 2nd we have printed the value at pointer “**\*(p+1)**”; 3rd we have printed the value at pointer “**\*(p+2)**”; 4th we have printed the value at pointer “**\*(p+3)**". This program will output the values at "0, 1, 2, 3" indices of an array "marks". The output of the following program is shown in figure 5.



***Figure 5: Pointer and Array Program 2 Output***

#### Code as described/written in the video

#include<iostream>

using namespace std;

int main(){

// Array Example

int marks[] = {23, 45, 56, 89};

int mathMarks[4];

mathMarks[0] = 2278;

mathMarks[1] = 738;

mathMarks[2] = 378;

mathMarks[3] = 578;

cout<<"These are math marks"<<endl;

cout<<mathMarks[0]<<endl;

cout<<mathMarks[1]<<endl;

cout<<mathMarks[2]<<endl;

cout<<mathMarks[3]<<endl;

// You can change the value of an array

marks[2] = 455;

cout<<"These are marks"<<endl;

// cout<<marks[0]<<endl;

// cout<<marks[1]<<endl;

// cout<<marks[2]<<endl;

// cout<<marks[3]<<endl;

for (int i = 0; i < 4; i++)

{

cout<<"The value of marks "<<i<<" is "<<marks[i]<<endl;

}

// Quick quiz: do the same using while and do-while loops?

// Pointers and arrays

int\* p = marks;

cout<<\*(p++)<<endl;

cout<<\*(++p)<<endl;

// cout<<"The value of \*p is "<<\*p<<endl;

// cout<<"The value of \*(p+1) is "<<\*(p+1)<<endl;

// cout<<"The value of \*(p+2) is "<<\*(p+2)<<endl;

// cout<<"The value of \*(p+3) is "<<\*(p+3)<<endl;

return 0;

}

# **Pointers vs References in C++**

[**Pointers**](https://www.geeksforgeeks.org/pointers-in-c-and-c-set-1-introduction-arithmetic-and-array/)**:** A pointer is a variable that holds memory address of another variable. A pointer needs to be dereferenced with **\*** operator to access the memory location it points to.

[**References**](https://www.geeksforgeeks.org/references-in-c/)**:** A reference variable is an alias, that is, another name for an already existing variable. A reference, like a pointer, is also implemented by storing the address of an object.   
A reference can be thought of as a constant pointer (not to be confused with a pointer to a constant value!) with automatic indirection, i.e the compiler will apply the **\***operator for you

### Structures, Unions & Enums in C++

**Structures in C++:**

The structure is a user-defined data type that is available in C++. Structures are used to combine different types of data types, just like an array is used to combine the same type of data types. An example program for creating a structure is shown in Code Snippet 1.

struct employee

{

/\* data \*/

int eId;

char favChar;

float salary;

};

***Code Snippet 1: Creating a Structure Program***

As shown in Code Snippet 1, we have created a structure with the name “employee”, in which we have declared three variables of different data types (eId, favchar, salary). As we have created a structure now we can create instances of our structure employee. An example program for creating instances of structure employees is shown in Code Snippet 2.

int main() {

struct employee harry;

harry.eId = 1;

harry.favChar = 'c';

harry.salary = 120000000;

cout<<"The value is "<<harry.eId<<endl;

cout<<"The value is "<<harry.favChar<<endl;

cout<<"The value is "<<harry.salary<<endl;

return 0;

}

***Code Snippet 2: Creating Structure instances***

As shown in Code Snippet 2, 1st we have created a structure variable “harry” of type “employee”, 2nd we have assigned values to (eId, favchar, salary) fields of the structure employee and at the end we have printed the value of “salary”.

Another way to create structure variables without using the keyword “struct” and the name of the struct is shown in Code Snippet 3.

typedef struct employee

{

/\* data \*/

int eId; //4

char favChar; //1

float salary; //4

} ep;

***Code Snippet 3: Creating Structure Program 2***

As shown in Code Snippet 3, we have used a keyword “**typedef**” before struct and after the closing bracket of structure, we have written “ep”. Now we can create structure variables without using the keyword “struct” and name of the struct. An example is shown in Code Snippet 4.

int main(){

ep harry;

struct employee shubham;

struct employee rohanDas;

harry.eId = 1;

harry.favChar = 'c';

harry.salary = 120000000;

cout<<"The value is "<<harry.eId<<endl;

cout<<"The value is "<<harry.favChar<<endl;

cout<<"The value is "<<harry.salary<<endl;

return 0;

}

***Code Snippet 4: Creating Structure instance 2***

As shown in Code Snippet 4, we have created a structure instance “harry” by just writing “ep” before it.

**Unions in C++:**

Unions are similar to structures but they provide better memory management then structures.  Unions use shared memory so only 1 variable can be used at a time. An example program to create unions is shown in Code Snippet 5.

union money

{

/\* data \*/

int rice; //4

char car; //1

float pounds; //4

};

***Code Snippet 5: Creating Unions Program***

As shown in Code Snippet 5, we have created a union with the name “money” in which we have declared three variables of different data types (rice, car, pound). The main thing to note here is that:

* We can only use 1 variable at a time otherwise the compiler will give us a garbage value
* The compiler chooses the data type which has maximum memory for the allocation.

An example program for creating an instance of union money is shown in Code Snippet 6.

int main(){

union money m1;

m1.rice = 34;

cout<<m1.rice;

return 0;

}

***Code Snippet 6: Creating a Union Instance***

As shown in Code Snippet 6, 1st we have created a union variable “m1” of type “money”, 2nd we have assigned values to (rice) fields of the union money, and in the end, we have printed the value of “rice”. The main thing to note here is that once we have assigned a value to the union field “rice”, now we cannot use other fields of the union otherwise we will get garbage value. The output for the following program is shown in figure 1.



***Figure 1: Creating Union Instance Output***

**Enums in C++:**

Enums are user-defined types which consist of named constants. Enums are used to make the program more readable. An example program for enums is shown in Code Snippet 8.

int main(){

enum Meal{ breakfast, lunch, dinner};

Meal m1 = lunch;

cout<<m1;

return 0;

}

***Code Snippet 7: Enums Program***

As shown in Code Snippet 7, 1st we have created an enum “Meal” in which we have stored three named constants (breakfast, lunch, dinner). 2nd we have assigned the value of “lunch” to the variable “m1” and at the end, we have printed “m1”. The main thing to note here is that (breakfast, lunch, dinner) are constants; the value for “breakfast” is “0”, the value for “lunch” is “1” and the value for “dinner” is “2”. The output for the following program is shown in figure 2.



***Figure 2: Enums Program Output***

**Code as described/written in the video:**

#include<iostream>

using namespace std;

typedef struct employee

{

/\* data \*/

int eId; //4

char favChar; //1

float salary; //4

} ep;

union money

{

/\* data \*/

int rice; //4

char car; //1

float pounds; //4

};

int main(){

enum Meal{ breakfast, lunch, dinner};

Meal m1 = lunch;

cout<<(m1==2);

// cout<<breakfast;

// cout<<lunch;

// cout<<dinner;

// union money m1;

// m1.rice = 34;

// m1.car = 'c';

// cout<<m1.car;

// ep harry;

// struct employee shubham;

// struct employee rohanDas;

// harry.eId = 1;

// harry.favChar = 'c';

// harry.salary = 120000000;

// cout<<"The value is "<<harry.eId<<endl;

// cout<<"The value is "<<harry.favChar<<endl;

// cout<<"The value is "<<harry.salary<<endl;

return 0;

}

# **C++ Functions**

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

You can pass data, known as parameters, into a function.

Functions are used to perform certain actions, and they are important for reusing code: Define the code once, and use it many times.

## **Create a Function:**

C++ provides some pre-defined functions, such as main(), which is used to execute code. But you can also create your own functions to perform certain actions.

To create (often referred to as declare) a function, specify the name of the function, followed by parentheses **()**:

A function is a set of statements that take inputs, do some specific computation and produces output.

The idea is to put some commonly or repeatedly done task together and make a function so that instead of writing the same code again and again for different inputs, we can call the function.

**Syntax:**

Return\_type Fun\_name(arg1, arg2,…..)

**Why do we need functions?**

* Functions help us in reducing code redundancy. If functionality is performed at multiple places in software, then rather than writing the same code, again and again, we create a function and call it everywhere. This also helps in maintenance as we have to change at one place if we make future changes to the functionality.
* Functions make code modular. Consider a big file having many lines of codes. It becomes really simple to read and use the code if the code is divided into functions.
* Functions provide abstraction. For example, we can use library functions without worrying about their internal working.

## **Function Declaration and Definition:**

A C++ function consist of two parts:

* **Declaration:** the function's name, return type, and parameters (if any)
* **Definition:** the body of the function (code to be executed)

**1.Function Declaration:**  
A function declaration tells the compiler about the number of parameters function takes, data-types of parameters and return type of function. Putting parameter names in function declaration is optional in the function declaration, but it is necessary to put them in the definition. Below are an example of function declarations. (parameter names are not there in below declarations)

-

-

-

-

-

-

-

-

#### //Code with HARRY

#### Functions in C++:

Functions are the main part of top-down structured programming. We break the code into small pieces and make functions of that code. Functions help us to reuse the code easily. An example program for the function is shown in Code Snippet 1.

int sum(int a, int b){

int c = a+b;

return c;

}

int main(){

int num1, num2;

cout<<"Enter first number"<<endl;

cin>>num1;

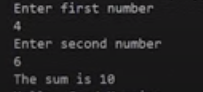
cout<<"Enter second number"<<endl;

cin>>num2;

cout<<"The sum is "<<sum(num1, num2);

return 0;

}



**Figure 1: Function Output**

#### Function Prototype in C++:

The function prototype is the template of the function which tells the details of the function e.g(name, parameters) to the compiler. Function prototypes help us to define a function after the function call. An example of a function prototype is shown in Code Snippet 3.

// Function prototype

int sum(int a, int b);

**Code Snippet 3: Function Prototype**

As shown in Code Snippet 3, we have made a function prototype of the function “sum”, this function prototype will tell the compiler that the function “sum” is declared somewhere in the program which takes two integer parameters and returns an integer value. Some examples of acceptable and not acceptable prototypes are shown below:

* **int sum(int a, int b); //Acceptable**
* **int sum(int a, b); // Not Acceptable**
* **int sum(int, int); //Acceptable**

##### **1.Formal Parameters:**

The variables which are declared in the function are called a formal parameter. For example, as shown in Code Snippet 1, the variables “a” and “b” are the formal parameters.

##### **2.Actual Parameters:**

The values which are passed to the function are called actual parameters. For example, as shown in Code Snippet 2, the variables “num1” and “num2” are the actual parameters.

The function doesn't need to have parameters or it should return some value. An example of the void function is shown in Code Snippet 4.

void g(){

cout<<"\nHello, Good Morning";

}

**Code Snippet 4: Void Function**

As shown in Code Snippet 4, void as a return type means that this function will not return anything, and this function has no parameters. Whenever we will call this function it will print “Hello, Good Morning”

### Call by Value & Call by Reference in C++:

#### Call by Value in C++:

Call by value is a method in C++ to pass the values to the function arguments. In case of call by value the copies of actual parameters are sent to the formal parameter, which means that if we change the values inside the function that will not affect the actual values. An example program for the call by value is shown in Code Snippet 1.

void swap(int a, int b){ //temp a b

int temp = a; //4 4 5

a = b; //4 5 5

b = temp; //4 5 4

}

**Code Snippet 1: Call by Value Swap Function**

As shown in Code Snippet 1, we created a swap function which is taking two parameters “int a” and “int b”. In function body values of the variable, “a” and “b” are swapped.  An example program is shown in Code Snippet 2, which calls the swap function and passes values to it.

int main(){

int x =4, y=5;

cout<<"The value of x is "<<x<<" and the value of y is "<<y<<endl;

swap(x, y);

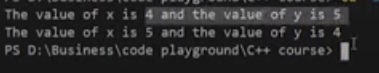
cout<<"The value of x is "<<x<<" and the value of y is "<<y<<endl;

return 0;

}

**Code Snippet 2: Passing Values to Swap Function**

As shown in Code Snippet 2, we have initialized two integer variables “a” and “b” and printed their values. Then we called a “swap” function and passed values of variables “a” and “b” and again printed the values of variables “a” and “b”. The output for the following program is shown in figure 1.



**Figure 1: Call by Value Swap Function Output**

As shown in figure 3, the values of “a” and “b” are the same for both times they are printed. So the main point here is that when the call by value method is used it doesn’t change the actual values because copies of actual values are sent to the function.

#### Call by Pointer in C++:

A call by the pointer is a method in C++ to pass the values to the function arguments. In the case of call by pointer, the address of actual parameters is sent to the formal parameter, which means that if we change the values inside the function that will affect the actual values. An example program for the call by reference is shown in Code Snippet 3.

// Call by reference using pointers

void swapPointer(int\* a, int\* b){ //temp a b

int temp = \*a; //4 4 5

\*a = \*b; //4 5 5

\*b = temp; //4 5 4

}

**Code Snippet 3: Call by Pointer Swap Function**

As shown in Code Snippet 3, we created a swap function which is taking two pointer parameters “int\* a” and “int\* b”. In function body values of pointer variables, “a” and “b” are swapped.  An example program is shown in Code Snippet 4, which calls the swap function and passes values to it.

int main(){

int x =4, y=5;

cout<<"The value of x is "<<x<<" and the value of y is "<<y<<endl;

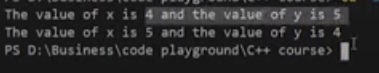
swapPointer(&x, &y); //This will swap a and b using pointer reference

cout<<"The value of x is "<<x<<" and the value of y is "<<y<<endl;

return 0;

**Code Snippet 4: Passing Values to Call by Pointer Swap Function**

As shown in Code Snippet 4, we have initialized two integer variables “a” and “b” and printed their values. Then we called a “swap” function and passed addresses of variables “a” and “b” and again printed the values of variables “a” and “b”. The output for the following program is shown in figure 2.



**Figure 2: Call by Pointer Swap Function Output**

As shown in figure 2, the values of “a” and “b” are swapped when the swap function is called. So the main point here is that when the call by pointer method is used it changes the actual values because addresses of actual values are sent to the function.

#### Call by Reference in C++:

Call by reference is a method in C++ to pass the values to the function arguments. In the case of call by reference, the reference of actual parameters is sent to the formal parameter, which means that if we change the values inside the function that will affect the actual values. An example program for a call by reference is shown in Code Snippet 5.

void swapReferenceVar(int &a, int &b){ //temp a b

int temp = a; //4 4 5

a = b; //4 5 5

b = temp; //4 5 4

}

**Code Snippet 5: Call by Reference Swap Function**

As shown in Code Snippet 5, we created a swap function that is taking reference of “int &a” and “int &b” as parameters. In function body values of variables, “a” and “b” are swapped.  An example program is shown in Code Snippet 6, which calls the swap function and passes values to it.

int main(){

int x =4, y=5;

cout<<"The value of x is "<<x<<" and the value of y is "<<y<<endl;

swapReferenceVar(x, y); //This will swap a and b using reference variables

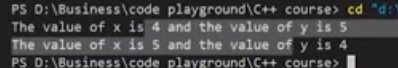
cout<<"The value of x is "<<x<<" and the value of y is "<<y<<endl;

return 0;

}

**Code Snippet 6: Passing Values to Call by Reference Swap Function**

As shown in Code Snippet 6, we have initialized two integer variables “a” and “b” and printed their values. Then we called a “swap” function and passed variables “a” and “b” and again printed the values of variables “a” and “b”. The output for the following program is shown in figure 3.



**Figure 3: Call by Reference Swap Function Output**

As shown in figure 3, the values of “a” and “b” are swapped when the swap function is called. So the main point here is that when the call by reference method is used it changes the actual values because references of actual values are sent to the function.

#### Inline Functions in C++:

Inline functions are used to reduce the function call. When one function is being called multiply times in the program it increases the execution time, so inline function is used to reduce time and increase program efficiency. If the inline function is being used when the function is called, the inline function expands the whole function code at the point of a function call, instead of running the function. Inline functions are considered to be used when the function is small otherwise it will not perform well. Inline is not recommended when static variables are being used in the function. An example of an inline function is shown in Code Snippet 1.

inline int product(int a, int b){

return a\*b;

}

**Code Snippet 1: Inline function**

As shown in Code Snippet 1, 1st inline keyword is used to make the function inline. 2nd a product function is created which has two arguments and returns the product of them. Now we will call the product function multiple times in our main program which is shown in Code Snippet 2.

int main(){

int a, b;

cout<<"Enter the value of a and b"<<endl;

cin>>a>>b;

cout<<"The product of a and b is "<<product(a,b)<<endl;

cout<<"The product of a and b is "<<product(a,b)<<endl;

cout<<"The product of a and b is "<<product(a,b)<<endl;

cout<<"The product of a and b is "<<product(a,b)<<endl;

cout<<"The product of a and b is "<<product(a,b)<<endl;

cout<<"The product of a and b is "<<product(a,b)<<endl;

cout<<"The product of a and b is "<<product(a,b)<<endl;

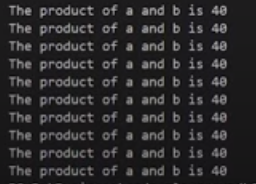
cout<<"The product of a and b is "<<product(a,b)<<endl;

return 0;

}

**Code Snippet 2: Calling Inline Product Function**

As shown in Code Snippet 2, we called the product function multiple times. The main thing to note here is that the function will not run instead of it the function code will be copied at the place where the function is being called. This will increase the execution time of the program because the compiler doesn’t have to copy the values and get the return value again and again from the compiler. The output of the following program is shown in figure 1.



**Figure 1: Inline Function Output**

#### Default Arguments in C++:

Default arguments are those values which are used by the function if we don’t input our value. It is recommended to write default arguments after the other arguments. An example program for default arguments is shown in Code Snippet 3.

float moneyReceived(int currentMoney, float factor=1.04){

return currentMoney \* factor;

}

int main(){

int money = 100000;

cout<<"If you have "<<money<<" Rs in your bank account, you will recive "<<moneyReceived(money)<< "Rs after 1 year"<<endl;

cout<<"For VIP: If you have "<<money<<" Rs in your bank account, you will recive "<<moneyReceived(money, 1.1)<< " Rs after 1 year";

return 0;

}

**Code Snippet 3: Default Argument Example Program**

As shown in Code Snippet 3, we created a “moneyReceived” function which has two arguments “int currentMoney” and “float factor=1.04”. This function returns the product of “currentMoney” and “factor”. In our main function, we called “moneyReceived” function and passed one argument “money”. Again we called “moneyReceived” function and passed two arguments ”money” and “1.1”. The main thing to note here is that when we passed only one argument “money” to the function at that time the default value of the argument “factor” will be used. But when we passed both arguments then the default value will not be used. The output for the following program is shown in figure 2.



**Figure 2: Default Argument Example Program Output**

#### Constant Arguments in C++:

Constant arguments are used when you don’t want your values to be changed or modified by the function. An example of constant arguments is shown in Code Snippet 4.

int strlen(const char \*p){

}

**Code Snippet 4: Constant Arguments Example**

As shown in Code Snippet 4, we created a “strlen” function which takes a constant argument “p”. As the argument is constant so its value won’t be modified.

#### Code as described/written in the video

#include<iostream>

using namespace std;

inline int product(int a, int b){

// Not recommended to use below lines with inline functions

// static int c=0; // This executes only once

// c = c + 1; // Next time this function is run, the value of c will be retained

return a\*b;

}

float moneyReceived(int currentMoney, float factor=1.04){

return currentMoney \* factor;

}

// int strlen(const char \*p){

// }

int main(){

int a, b;

// cout<<"Enter the value of a and b"<<endl;

// cin>>a>>b;

// cout<<"The product of a and b is "<<product(a,b)<<endl;

int money = 100000;

cout<<"If you have "<<money<<" Rs in your bank account, you will recive "<<moneyReceived(money)<< "Rs after 1 year"<<endl;

cout<<"For VIP: If you have "<<money<<" Rs in your bank account, you will recive "<<moneyReceived(money, 1.1)<< " Rs after 1 year";

return 0;

}

**Recursion and Recursive Function:**

When a function calls itself it is called recursion and the function which is calling itself is called a recursive function. The recursive function consists of a base case and recursive condition. It is very important to add a base case in recursive function otherwise recursive function will never stop executing. An example of the recursive function is shown in Code Snippet 1.

int factorial(int n){

if (n<=1){

return 1;

}

return n \* factorial(n-1);

}

***Code Snippet 1: Factorial Recursive Function***

As shown in Code Snippet 1, we created a “factorial” function which takes one argument. In the function body, there is a base case which checks that if the value of variable “n” is smaller or equal to “1” if the condition is “true” return “1”. And there is a recursive condition that divides the bigger value to smaller values and at the end returns a factorial. These are the steps which will be performed by recursive condition:

* **4 \* factorial( 4-1 )**
* **4 \* 3 \* factorial( 3-1 )**
* **4\* 3 \* 2 \* factorial( 2-1 )**
* **4 \* 3 \* 2 \* 1**

An example to pass the value to the recursive factorial function is shown in Code Snippet 2.

int main(){

int a;

cout<<"Enter a number"<<endl;

cin>>a;

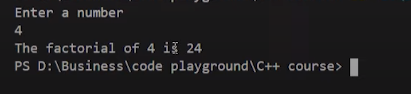
cout<<"The factorial of "<<a<< " is "<<factorial(a)<<endl;

return 0;

}

***Code Snippet 2: Factorial Recursive Function Call***

As shown in Code Snippet 2, we created an integer variable “a”, which takes input at the runtime and that value is passed to the factorial function. The output for the following program is shown in figure 1.



***Figure 1: Factorial Recursive Function Output***

As shown in figure 1, we input the value “4” and it gives us the factorial of it which is “24”. Another example of a recursive function for the Fibonacci series is shown in Code Snippet 3.

int fib(int n){

if(n<2){

return 1;

}

return fib(n-2) + fib(n-1);

}

***Code Snippet 3: Fibonacci Recursive Function***

As shown in Code Snippet 3, we created a “fib” function which takes one argument. In the function body, there is a base case which checks that if the value of variable “n” is smaller than “2”, if the condition is “true” return “1”. And there is a recursive condition that divides the bigger value to smaller values and at the end returns a Fibonacci number. An example to pass the value to the Fibonacci function is shown in Code Snippet 4.

int main(){

int a;

cout<<"Enter a number"<<endl;

cin>>a;

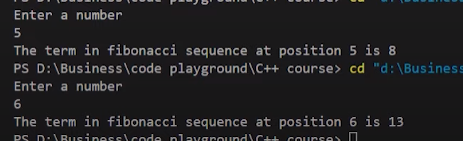
cout<<"The term in fibonacci sequence at position "<<a<< " is "<<fib(a)<<endl;

return 0;

}

***Code Snippet 4: Fibonacci Recursive Function Call***

As shown in Code Snippet 4, we created an integer variable “a”, which takes input at the runtime and that value is passed to the Fibonacci function. The output for the following program is shown in figure 2.



***Figure 2: Fibonacci Recursive Function Output***

As shown in figure 2, 1st we input the value “5” and it gives us the Fibonacci number at that place which is “8”. 2nd we input the value “6” and it gives us the Fibonacci number at that place which is “13”.

One thing to note here is that recursive functions are not always the best option. They perform well in some problems but not in every problem.

#### Code as described/written in the video

#include<iostream>

using namespace std;

int fib(int n){

if(n<2){

return 1;

}

return fib(n-2) + fib(n-1);

}

// fib(5)

// fib(4) + fib(3)

// fib(2) + fib(3) + fib(2) + fib(3)

int factorial(int n){

if (n<=1){

return 1;

}

return n \* factorial(n-1);

}

// Step by step calculation of factorial(4)

// factorial(4) = 4 \* factorial(3);

// factorial(4) = 4 \* 3 \* factorial(2);

// factorial(4) = 4 \* 3 \* 2 \* factorial(1);

// factorial(4) = 4 \* 3 \* 2 \* 1;

// factorial(4) = 24;

int main(){

// Factorial of a number:

// 6! = 6\*5\*4\*3\*2\*1 = 720

// 0! = 1 by definition

// 1! = 1 by definition

// n! = n \* (n-1)!

int a;

cout<<"Enter a number"<<endl;

cin>>a;

// cout<<"The factorial of "<<a<< " is "<<factorial(a)<<endl;

cout<<"The term in fibonacci sequence at position "<<a<< " is "<<fib(a)<<endl;

return 0;

}

#### Function Overloading in C++:

Function overloading is a process to make more than one function with the same name but different parameters, numbers, or sequence. An example program to explain function overloading is shown in Code Snippet 1.

int sum(float a, int b){

cout<<"Using function with 2 arguments"<<endl;

return a+b;

}

int sum(int a, int b, int c){

cout<<"Using function with 3 arguments"<<endl;

return a+b+c;

}

**Code Snippet 1: Sum Function Overloading Example**

int main(){

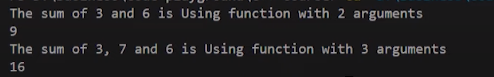
cout<<"The sum of 3 and 6 is "<<sum(3,6)<<endl;

cout<<"The sum of 3, 7 and 6 is "<<sum(3, 7, 6)<<endl;

return 0;

}

**Code Snippet 2: Sum Function Call**



**Figure 1: Sum Function Output**

Another example of function overloading is shown in Code Snippet 3.

// Calculate the volume of a cylinder

int volume(double r, int h){

return(3.14 \* r \*r \*h);

}

// Calculate the volume of a cube

int volume(int a){

return (a \* a \* a);

}

// Rectangular box

int volume (int l, int b, int h){

return (l\*b\*h);

}

**Code Snippet 3: Volume Function Overloading Example**

int main(){

cout<<"The volume of cuboid of 3, 7 and 6 is "<<volume(3, 7, 6)<<endl;

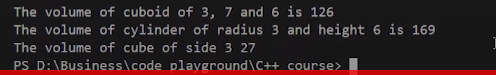
cout<<"The volume of cylinder of radius 3 and height 6 is "<<volume(3, 6)<<endl;

cout<<"The volume of cube of side 3 is "<<volume(3)<<endl;

return 0;

}

**Code Snippet 4: Volume Function Call**



**Figure 2: Volume Function Output**

As shown in figure 2, all three “volume” functions run fine and give us the required output.

#### Code as described/written in the video

#include<iostream>

using namespace std;

int sum(float a, int b){

cout<<"Using function with 2 arguments"<<endl;

return a+b;

}

int sum(int a, int b, int c){

cout<<"Using function with 3 arguments"<<endl;

return a+b+c;

}

// Calculate the volume of a cylinder

int volume(double r, int h){

return(3.14 \* r \*r \*h);

}

// Calculate the volume of a cube

int volume(int a){

return (a \* a \* a);

}

// Rectangular box

int volume (int l, int b, int h){

return (l\*b\*h);

}

int main(){

cout<<"The sum of 3 and 6 is "<<sum(3,6)<<endl;

cout<<"The sum of 3, 7 and 6 is "<<sum(3, 7, 6)<<endl;

cout<<"The volume of cuboid of 3, 7 and 6 is "<<volume(3, 7, 6)<<endl;

cout<<"The volume of cylinder of radius 3 and height 6 is "<<volume(3, 6)<<endl;

cout<<"The volume of cube of side 3 is "<<volume(3)<<endl;

return 0;

}

### Object Oriented Programming in C++

#### Why Object-Oriented Programming ???

Before we discuss object-oriented programming, we need to learn why we need object-oriented programming?

* C++ language was designed with the main intention of adding object-oriented programming to C language
* As the size of the program increases readability, maintainability, and bug-free nature of the program decrease.
* This was the major problem with languages like C which relied upon functions or procedure (hence the name procedural programming language)
* As a result, the possibility of not addressing the problem adequately was high
* Also, data was almost neglected, data security was easily compromised
* Using classes solves this problem by modeling program as a real-world scenario

#### Difference between Procedure Oriented Programming and Object-Oriented Programming:

##### **1.Procedure Oriented Programming:**

* Consists of writing a set of instruction for the computer to follow
* The main focus is on functions and not on the flow of data
* Functions can either use local or global data
* Data moves openly from function to function

##### **2.Object-Oriented Programming:**

* Works on the concept of classes and object
* A class is a template to create objects
* Treats data as a critical element
* Decomposes the problem in objects and builds data and functions around the objects

#### Basic Concepts in Object-Oriented Programming:

* **Classes -**Basic template for creating objects.
* **Objects –**Basic run-time entities
* **Data Abstraction & Encapsulation –**Wrapping data and functions into a single unit
* **Inheritance –**Properties of one class can be inherited into others
* **Polymorphism –**Ability to take more than one forms
* **Dynamic Binding –**Code which will execute is not known until the program runs
* **Message Passing –**message (Information) call format

#### Benefits of Object-Oriented Programming:

* Better code reusability using objects and inheritance
* Principle of data hiding helps build secure systems
* Multiple Objects can co-exist without any interference
* Software complexity can be easily managed

### Classes, Public and Private access modifiers in C++

#### Why use classes instead of structures ???

Classes and structures are somewhat the same but still, they have some differences. For example, we cannot hide data in structures which means that everything is public and can be accessed easily which is a major drawback of the structure because structures cannot be used where data security is a major concern. Another drawback of structures is that we cannot add functions in it.

#### Classes in C++:

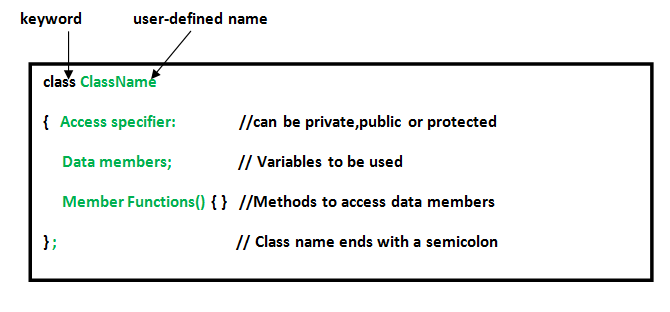
Classes are user-defined data-types and are a template for creating objects. Classes consist of variables and functions which are also called class members.

**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.**For Example: Consider the Class of **Cars**. There may be many cars with different names and brand but all of them will share some common properties like all of them will have 4 wheels, Speed Limit, Mileage range etc. So here, Car is the class and wheels, speed limits, mileage are their properties.

* A Class is a user defined data-type which has data members and member functions.
* Data members are the data variables and member functions are the functions used to manipulate these variables and together these data members and member functions defines the properties and behaviour of the objects in a Class.
* In the above example of class Car, the data member will be speed limit, mileage etc and member functions can be apply brakes, increase speed etc.

**An Object** **is an instance of a Class. When a class is defined, no memory is allocated but when it is instantiated (i.e. an object is created) memory is allocated.**

**Defining Class and Declaring Objects:**

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;**

**Accessing data members and member functions**: The data members and member functions of class can be accessed using the dot(‘.’) operator with the object. For example if the name of object is obj and you want to access the member function with the name printName() then you will have to write obj.printName() .

**Accessing Data Members**

The public data members are also accessed in the same way given however the private data members are not allowed to be accessed directly by the object. Accessing a data member depends solely on the access control of that data member.  
This access control is given by [Access modifiers in C++](https://www.geeksforgeeks.org/access-modifiers-in-c/). There are three access modifiers : **public, private and protected**.

There are 3 types of access modifiers available in C++:

1. **Public**
2. **Private**
3. **Protected**

**Note**: **If we do not specify any access modifiers for the members inside the class then by default the access modifier for the members will be Private.**

Let us now look at each one these access modifiers in details:

 **1. Public**: All the class members declared under the public specifier will be available to everyone. The data members and member functions declared as public can be accessed by other classes and functions too. The public members of a class can be accessed from anywhere in the program using the direct member access operator (.) with the object of that class.

**2.Private**: The class members declared as *private* can be accessed only by the member functions inside the class. They are not allowed to be accessed directly by any object or function outside the class. Only the member functions or the [friend functions](https://www.geeksforgeeks.org/friend-class-function-cpp/) are allowed to access the private data members of a class.

**3. Protected**: Protected access modifier is similar to private access modifier in the sense that it can’t be accessed outside of it’s class unless with the help of friend class, the difference is that the class members declared as Protected can be accessed by any subclass(derived class) of that class as well.

**Note**: This access through inheritance can alter the access modifier of the elements of base class in derived class depending on the [modes of Inheritance](https://www.geeksforgeeks.org/inheritance-in-c/#Modes%20of%20Inheritance).

#### Code as described/written in the video

#include<iostream>

using namespace std;

class Employee

{

private:

int a, b, c;

public:

int d, e;

void setData(int a1, int b1, int c1); // Declaration

void getData(){

cout<<"The value of a is "<<a<<endl;

cout<<"The value of b is "<<b<<endl;

cout<<"The value of c is "<<c<<endl;

cout<<"The value of d is "<<d<<endl;

cout<<"The value of e is "<<e<<endl;

}

};

void Employee :: setData(int a1, int b1, int c1){

a = a1;

b = b1;

c = c1;

}

int main(){

Employee harry;

// harry.a = 134; -->This will throw error as a is private

harry.d = 34;

harry.e = 89;

harry.setData(1,2,4);

harry.getData();

return 0;}

**Constructors:**

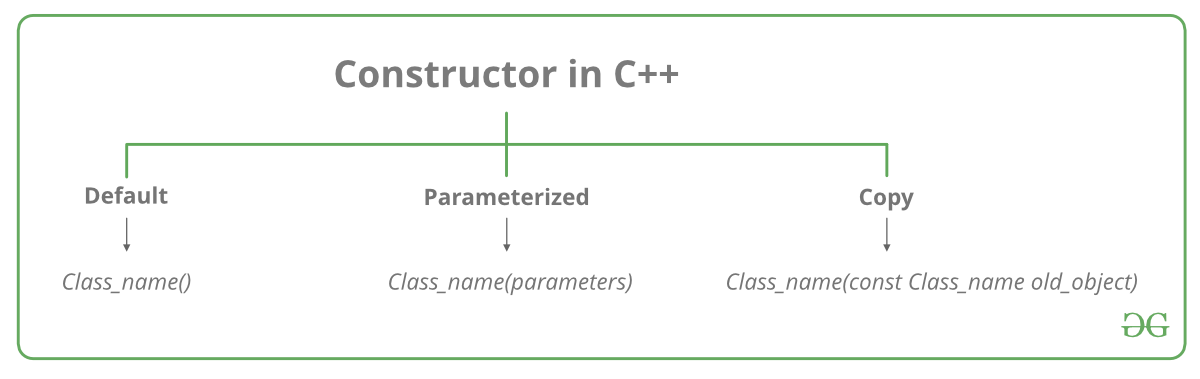
**What is constructor?**

A constructor is a special type of member function of a class which initializes objects of a class. In C++, Constructor is automatically called when object(instance of class) create. It is special member function of the class because it does not have any return type.

**How constructors are different from a normal member function?**

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.
* It must be placed in public section of class.
* If we do not specify a constructor, C++ compiler generates a default constructor for object (expects no parameters and has an empty body).



**Types of Constructors:**

1.[**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.

**Note:**Even if we do not define any constructor explicitly, the compiler will automatically provide a default constructor implicitly.

**2.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.

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:**
  1. It is used to initialize the various data elements of different objects with different values when they are created.
  2. It is used to overload constructors.
* **Can we have more than one constructor in a class?**  
         Yes, It is called [Constructor Overloading](https://www.geeksforgeeks.org/constructor-overloading-c/).

**3.Copy Constructor:**

**What is a copy constructor?**  
A copy constructor is a member function that initializes an object using another object of the same class. A copy constructor has the following

**general function prototype:**

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

**When is** **copy constructor called?**   
In C++, a Copy Constructor may be called in the following cases:   
1. When an object of the class is returned by value.   
2. When an object of the class is passed (to a function) by value as an argument.   
3. When an object is constructed based on another object of the same class.   
4. When the compiler generates a temporary object.  
It is, however, not guaranteed that a copy constructor will be called in all these cases, because the C++ Standard allows the compiler to optimize the copy away in certain cases, one example is the [return value optimization (sometimes referred to as RVO)](http://en.wikipedia.org/wiki/Return_value_optimization).

# **Constructor Overloading in C++**

In C++, We can have more than one constructor in a class with same name, as long as each has a different list of arguments. This concept is known as Constructor Overloading and is quite similar to [function overloading](https://www.geeksforgeeks.org/function-overloading-c/). 

* Overloaded constructors essentially have the same name (exact name of the class) and differ by number and type of arguments.
* A constructor is called depending upon the number and type of arguments passed.
* While creating the object, arguments must be passed to let compiler know, which constructor needs to be called.

# **Destructors in C++**

**What is destructor?**   
Destructor is an instance member function which is invoked automatically whenever an object is going to destroy. Means, a destructor is the last function that is going to be called before an object is destroyed.

The thing to be noted is that destructor doesn’t destroys an object.

Destructors are usually **used to deallocate memory and do other cleanup for a class object** and its class members when the object is destroyed. A destructor is called for a class object when that object passes out of scope or is explicitly deleted.

**Syntax: ~**constructor-name();

**Properties of Destructor:**

* Destructor function is automatically invoked when the objects are destroyed.
* It cannot be declared static or const.
* The destructor does not have arguments.
* It has no return type not even void.
* An object of a class with a Destructor cannot become a member of the union.
* A destructor should be declared in the public section of the class.
* The programmer cannot access the address of destructor.

**When is destructor called?**   
A destructor function is called automatically when the object goes out of scope:   
(1) the function ends   
(2) the program ends   
(3) a block containing local variables ends   
(4) a delete operator is called

**How destructors are different from a normal member function?**   
Destructors have same name as the class preceded by a tilde (~)   
Destructors don’t take any argument and don’t return anything

**Nesting of members:**

#### Code as described/written in the video

// OOPs - Classes and objects

// C++ --> initially called --> C with classes by stroustroup

// class --> extension of structures (in C)

// structures had limitations

// - members are public

// - No methods

// classes --> structures + more

// classes --> can have methods and properties

// classes --> can make few members as private & few as public

// structures in C++ are typedefed

// you can declare objects along with the class declarion like this:

/\* class Employee{

// Class definition

} harry, rohan, lovish; \*/

// harry.salary = 8 makes no sense if salary is private

// Nesting of member functions

#include <iostream>

#include <string>

using namespace std;

class binary

{

private:

string s;

void chk\_bin(void);

public:

void read(void);

void ones\_compliment(void);

void display(void);

};

void binary::read(void)

{

cout << "Enter a binary number" << endl;

cin >> s;

}

void binary::chk\_bin(void)

{

for (int i = 0; i < s.length(); i++)

{

if (s.at(i) != '0' && s.at(i) != '1')

{

cout << "Incorrect binary format" << endl;

exit(0);

}

}

}

void binary::ones\_compliment(void)

{

chk\_bin();

for (int i = 0; i < s.length(); i++)

{

if (s.at(i) == '0')

{

s.at(i) = '1';

}

else

{

s.at(i) = '0';

}

}

}

void binary::display(void)

{

cout<<"Displaying your binary number"<<endl;

for (int i = 0; i < s.length(); i++)

{

cout << s.at(i);

}

cout<<endl;

}

int main()

{

binary b;

b.read();

// b.chk\_bin();

b.display();

b.ones\_compliment();

b.display();

return 0;

}

**Static Data Members in C++**

When a static data member is created, there is only a single copy of the data member which is shared between all the objects of the class. As we have discussed in our previous lecture that if the data members are not static then every object has an individual copy of the data member and it is not shared.

**Static Methods in C++:**

When a static method is created, they become independent of any object and class. Static methods can only access static data members and static methods. Static methods can only be accessed using the scope resolution operator. An example program is shown below to demonstrate static data members and static methods in C++.

class Employee

{

int id;

static int count;

public:

void setData(void)

{

cout << "Enter the id" << endl;

cin >> id;

count++;

}

void getData(void)

{

cout << "The id of this employee is " << id << " and this is employee number " << count << endl;

}

static void getCount(void){

// cout<<id; // throws an error

cout<<"The value of count is "<<count<<endl;

}

};

***Code Snippet 1: Employee Class***

// Count is the static data member of class Employee

int Employee::count; // Default value is 0

int main()

{

Employee harry, rohan, lovish;

// harry.id = 1;

// harry.count=1; // cannot do this as id and count are private

harry.setData();

harry.getData();

Employee::getCount();

rohan.setData();

rohan.getData();

Employee::getCount();

lovish.setData();

lovish.getData();

Employee::getCount();

return 0;

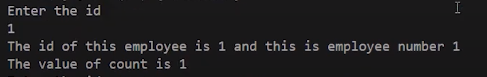
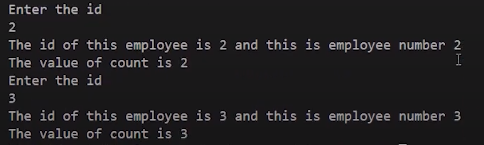
}

***Code Snippet 2: main Program***

As shown in Code Snippet 2:

* The count variable is declared whose default value is “0”.
* Then we created objects “harry”, “rohan”, and “lovish” of the employee data type
* The functions “setData”, “getData” are called by the object “harry”, the function “getCount” is called by using class name and scope resolution operator because it is a static method.
* The functions “setData”, “getData” are called by the object “rohan”, the function “getCount” is called by using class name and scope resolution operator because it is a static method.
* The functions “setData”, “getData” are called by the object “lovish”, the function “getCount” is called by using class name and scope resolution operator because it is a static method.

The output of the following program is shown in figures 1 and 2.

******

**Friend Function in C++**

Friend functions are those functions that have the right to access the private data members of class even though they are not defined inside the class. It is necessary to write the prototype of the friend function. One main thing to note here is that if we have written the prototype for the friend function in the class it will not make that function a member of the class. An example program to demonstrate the concept of friend function is shown below.

class Complex{

int a, b;

public:

void setNumber(int n1, int n2){

a = n1;

b = n2;

}

// Below line means that non member - sumComplex funtion is allowed to do anything with my private parts (members)

friend Complex sumComplex(Complex o1, Complex o2);

void printNumber(){

cout<<"Your number is "<<a<<" + "<<b<<"i"<<endl;

}

};

Complex sumComplex(Complex o1, Complex o2){

Complex o3;

o3.setNumber((o1.a + o2.a), (o1.b+o2.b))

;

return o3;

}

***Code Snippet 1: Complex Class***

We have defined a “sumComplex” friend function. In this function, the object “o3” is created which calls the “setNumber” function and passes the values of two objects after performing addition on them.

int main(){

Complex c1, c2, sum;

c1.setNumber(1, 4);

c1.printNumber();

c2.setNumber(5, 8);

c2.printNumber();

sum = sumComplex(c1, c2);

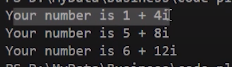
sum.printNumber();

return 0;

}

***Code Snippet 2: main Program***

The output of the following program is shown in figure 1.



***Figure 1: Complex Program Output***

**Properties of Friend Function:**

* Not in the scope of the class.
* Since it is not in the scope of the class, it cannot be called from the object of that class, for example, **sumComplex()**is invalid.
* A friend function can be invoked without the help of any object.
* Usually contain objects as arguments.
* Can be declared under the public or private access modifier, it will not make any difference.
* It cannot access the members directly by their names, it needs (object\_name.member\_name) to access any member.

**Inheritance in C++**

* Reusability is a very important feature of OOPs
* In C++ we can reuse a class and add additional features to it
* Reusing classes saves time and money
* Reusing already tested and debugged classes will save a lot of effort of developing and debugging the same thing again

**What is Inheritance in C++?**

* The concept of reusability in C++ is supported using inheritance
* We can reuse the properties of an existing class by inheriting it
* The existing class is called a base class
* The new class which is inherited from the base class is called a derived class
* Reusing classes saves time and money
* There are different types of inheritance in C++

**Forms of Inheritance in C++:**

* Single Inheritance
* Multiple Inheritance
* Hierarchical Inheritance
* Multilevel Inheritance
* Hybrid Inheritance

**1. Single Inheritance in C++:**

Single inheritance is a type of inheritance in which a derived class is inherited with only one base class. For example, we have two classes “employee” and “programmer”. If the “programmer” class is inherited from the “employee” class which means that the “programmer” class can now implement the functionalities of the “employee” class.

**2. Multiple Inheritances in C++:**

 Multiple inheritances are a type of inheritance in which one derived class is inherited with more than one base class. For example, we have three classes “employee”, “assistant” and “programmer”. If the “programmer” class is inherited from the “employee” and “assistant” class which means that the “programmer” class can now implement the functionalities of the “employee” and “assistant” class.

**3. Hierarchical Inheritance:**

A hierarchical inheritance is a type of inheritance in which several derived classes are inherited from a single base class. For example, we have three classes “employee”, “manager” and “programmer”. If the “programmer” and “manager” classes are inherited from the “employee” class which means that the “programmer” and “manager” class can now implement the functionalities of the “employee” class.

**4. Multilevel Inheritance in C++**

Multilevel inheritance is a type of inheritance in which one derived class is inherited from another derived class. For example, we have three classes “animal”, “mammal” and “cow”. If the “mammal” class is inherited from the “animal” class and “cow” class is inherited from “mammal” which means that the “mammal” class can now implement the functionalities of “animal” and “cow” class can now implement the functionalities of “mammal” class.

**5. Hybrid Inheritance in C++**

Hybrid inheritance is a combination of multiple inheritance and multilevel inheritance. In hybrid inheritance, a class is derived from two classes as in multiple inheritances. However, one of the parent classes is not a base class. For example, we have four classes “animal”, “mammal”, “bird”, and “bat”. If “mammal”  and “bird” classes are inherited from the “animal” class and “bat” class is inherited from “mammal” and “bird” classes which means that “mammal” and “bird” classes can now implement the functionalities of “animal” class and “bat” class can now implement the functionalities of “mammal” and “bird” classes

**Protected Access Modifiers in C++**

Protected access modifiers are similar to the private access modifiers but protected access modifiers can be accessed in the derived class whereas private access modifiers cannot be accessed in the derived class. A table is shown below which shows the behavior of access modifiers when they are derived “public”, “private”, and “protected”.

|  |  |  |  |
| --- | --- | --- | --- |
|  | **Public Derivation** | **Private Derivation** | **Protected Derivation** |
| **Private members** | Not Inherited | Not Inherited | Not Inherited |
| **Protected members** | Protected | Private | Protected |
| **Public members** | Public | Private | Protected |

As shown in the table,

1. If the class is inherited in public mode then its private members cannot be inherited in child class.
2. If the class is inherited in public mode then its protected members are protected and can be accessed in child class.
3. If the class is inherited in public mode then its public members are public and can be accessed inside child class and outside the class.
4. If the class is inherited in private mode then its private members cannot be inherited in child class.
5. If the class is inherited in private mode then its protected members are private and cannot be accessed in child class.
6. If the class is inherited in private mode then its public members are private and cannot be accessed in child class.
7. If the class is inherited in protected mode then its private members cannot be inherited in child class.
8. If the class is inherited in protected mode then its protected members are protected and can be accessed in child class.
9. If the class is inherited in protected mode then its public members are protected and can be accessed in child class.

**Constructors in Derived Class in C++:**

* We can use constructors in derived classes in C++
* If the base class constructor does not have any arguments, there is no need for any constructor in the derived class
* But if there are one or more arguments in the base class constructor, derived class need to pass argument to the base class constructor
* If both base and derived classes have constructors, base class constructor is executed first.

**Constructors in Multiple Inheritances:**

* In multiple inheritances, base classes are constructed in the order in which they appear in the class deceleration. For example if there are three classes “A”, “B”, and “C”, and the class “C” is inheriting classes “A” and “B”. If the class “A” is written before class “B” then the constructor of class “A” will be executed first. But if the class “B” is written before class “A” then the constructor of class “B” will be executed first.
* In multilevel inheritance, the constructors are executed in the order of inheritance. For example if there are three classes “A”, “B”, and “C”, and the class “B” is inheriting classes “A” and the class “C” is inheriting classes “B”. Then the constructor will run according to the order of inheritance such as the constructor of class “A” will be called first then the constructor of class “B” will be called and at the end constructor of class “C” will be called.

**Special Syntax:**

* C++ supports a special syntax for passing arguments to multiple base classes
* The constructor of the derived class receives all the arguments at once and then will pass the call to the respective base classes
* The body is called after the constructors is finished executing

**Syntax Example:**

Derived-Constructor (arg1, arg2, arg3….): Base 1-Constructor (arg1,arg2), Base 2-Constructor(arg3,arg4)

{

….

} Base 1-Constructor (arg1,arg2)

**Special Case of Virtual Base Class:**

* The constructors for virtual base classes are invoked before a non-virtual base class
* If there are multiple virtual base classes, they are invoked in the order declared
* Any non-virtual base class are then constructed before the derived class constructor is executed

**Polymorphism in C++**

“Poly” means several and “morphism” means form. So we can say that polymorphism is something that has several forms or we can say it as one name and multiple forms. There are two types of polymorphism:

* Compile-time polymorphism
* Run time polymorphism

**1.Compile Time Polymorphism:**

In compile-time polymorphism, it is already known which function will run. Compile-time polymorphism is also called **early binding**, which means that you are already bound to the function call and you know that this function is going to run. There are two types of compile-time polymorphism:

1. **Function Overloading:**

This is a feature that lets us create more than one function and the functions have the same names but their parameters need to be different. If function overloading is done in the program and function calls are made the compiler already knows that which functions to execute.

1. **Operator Overloading:**

This is a feature that lets us define operators working for some specific tasks. For example, we can overload the operator “+” and define its functionality to add two strings. Operator loading is also an example of compile-time polymorphism because the compiler already knows at the compile time which operator has to perform the task.

**2. Run Time Polymorphism:**

In the run-time polymorphism, the compiler doesn’t know already what will happen at run time. Run time polymorphism is also called **late binding.** The run time polymorphism is considered slow because function calls are decided at run time. Run time polymorphism can be achieved from the virtual function.

1. **Virtual Function:**

A function that is in the parent class but redefined in the child class is called a virtual function. “virtual” keyword is used to declare a virtual function.

**File I/O in C++: Working with Files :**

The file is a patent of data which is stored in the disk. Anything written inside the file is called a patent, for example: “**#include**” is a patent. The text file is the combination of multiple types of characters, for example, semicolon “;” is a character.

The computer read these characters in the file with the help of the ASCII code. Every character is mapped on some decimal number. For example, ASCII code for the character “A” is “65” which is a decimal number. These decimal numbers are converted into a binary number to make them readable for the computer because the computer can only understand the language of “0” and “1”.

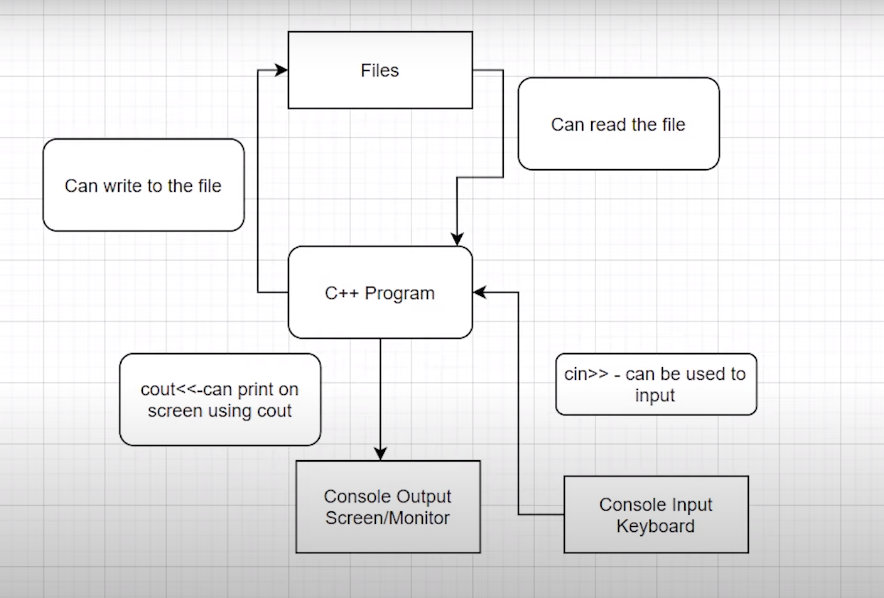
The reason that computers can only understand binary numbers is that a computer is made up of switches and switches only perform two operations “true” or “false”.

**File Input and Output in C++:**

The file can be of any type whether it is a file of a C++ program, file of a game, or any other type of file. There are two main operations which can be performed on files

* **Read File**
* **Write File**

An image is shown below to show the process of file read and write.



***Figure 1: File Read and Write Diagram***

As shown in figure 1,

1. The user can provide input to the C++ program by using keyboard through “cin>>” keyword
2. The user can get output from the C++ program on the monitor through “cout<<” keyword
3. The user can write on the file
4. The user can read the file

**File I/O in C++: Reading and Writing Files:**

These are some useful classes for working with files in C++

* fstreambase
* ifstream --> derived from fstreambase
* ofstream --> derived from fstreambase

In order to work with files in C++, you will have to open it. Primarily, there are 2 ways to open a file:

* Using the constructor
* Using the member function open() of the class

### File I/O in C++: Read/Write in the Same Program & Closing Files:

In this tutorial, we’ll learn about creating a program that will read from a file and write to the file in the same program using a constructor.

Before jumping on to the main thing, we’ll first give ourselves a quick revision of the things we had learned previously.

We had learned about the three most useful classes when we talk about File I/O, namely,

1. fstreambase
2. ifstream
3. ofstream.

All the above three classes can be used in a program by first including the header file, fstream.

**Reading File Operation Output:**

We learnt reading from a file using ifstream. Below snippet will help you recollect the same.

string st;

// Opening files using constructor and reading it

ifstream in("this.txt"); // Read operation

in>>st;

**Writing File Operation Output:**

We learnt reading from a file using ofstream. Below snippet will help you recollect the same.

string st = "Harry bhai";

// Opening files using constructor and writing it

ofstream out("this.txt"); // Write operation

out<<st;

Let me make these codes functional in the same program for you to easily understand the workflow.

Suppose we have a file named sample60.txt in the same directory, we can easily call the file infinite number of times in the same program only by maintaining different connections for different purposes, using

<object\_name>.close();

Now, let’s move on to our systems. Open your editors as well. Don’t forget to include the header file, <fstream>.

Follow these steps below to first write into the empty file:

1. Create a text file “sample60.txt” in the same directory as that of the program.
2. Create a string variable name.
3. Create an object hout(name it whatever you wish) using ofstream passing the text file, sample60.txt into it. This establishes a connection between your program and the text file.
4. Take input from the user using cin into the name string.(You can write manually as well)
5. Pass this name string to the object hout. The string name gets written in the text  file.
6. Disconnect the file with the program since we are done writing to it using hout.close().

Since the file has been disconnected from the program, we can connect it again for any other purpose in the same program independently.

Follow these steps below to read from the file we just wrote into:

1. Create a string variable content.
2. Create an object hin(name it whatever you wish) using ifstream passing the text file, sample60.txt into it. This establishes a new connection between your program and the text file.
3. Fill in the string using the object hin. (Use getline, which we talked about in the last video, to take into input the whole line from the text file.)
4. Give output to the user, the string we filled in with the content in the text file.
5. Disconnect the file with the program since we are done reading from it using hin.close().

#include<iostream>

#include<fstream>

using namespace std;

int main(){

// connecting our file with hout stream

ofstream hout("sample60.txt");

// creating a name string variable and filling it with string entered by the user

string name;

cout<<"Enter your name: ";

cin>>name;

// writing a string to the file

hout<<name + " is my name";

// disconnecting our file

hout.close();

// connecting our file with hin stream

ifstream hin("sample60.txt");

// creating a content string variable and filling it with string present there in the text file

string content;

hin>>content;

cout<<"The content of the file is: "<<content;

// disconnecting our file

hin.close();

return 0;

}

Let’s run the program we just created, The output will look like this:

Enter your name: Harry

The content of the file is: Harry

So when we input a string “Harry” into the text file, it gets written there in the file as below, and when we read it from the file, it gives output as below. Since we used hin and not getline, it could read just the first word.

The content of the file is: Harry

**File I/O in C++: open() and eof() functions:**

In this tutorial, we are going to learn about the member functions open and eof of the objects we learnt about previously.

I remember teaching you all about the two methods to open a text file in our C++ program, first one using a constructor which we discussed in the last tutorial, and the second one, using the member function open, which is to be dealt with today.

**Using the member function open:**

The member function open is used to connect the text file to the C++ program when passed into it.

Understanding the snippet below:

1. Unlike what we did earlier passing the text file in the object while creating it, we’ll first just declare an object out(any name you wish) of the type ofstream and use its open method to open the text file in the program.
2. We’ll pass some string lines to the text file using the out operation.
3. We’ll now close the file using the close function. Now closing is explicitly used to make the system know that we are done with the file. It is always good to use this.

This was all about writing to a file. We’ll now move to the eof function’s vitality in File I/O.

#include <iostream>

#include <fstream>

using namespace std;

int main()

{

// declaring an object of the type ofstream

ofstream out;

//connecting the object out to the text file using the member function open()

out.open("sample60.txt");

//writing to the file

out <<"This is me\n";

out <<"This is also me";

//closing the file connection

out.close();

return 0;

}

Copy

**Using the member function eof:**

The member function eof(End-of-file) returns a boolean true if the file reaches the end of it and false if not.

Understanding the snippet below:

1. We’ll first declare an object in(any name you wish) of the type ifstream and use its open method similar to what we did above, to open the text file in the program.
2. And now, we’ll declare the string variable st to store the content we’ll receive from the text file sample60.txt.
3. Now since we not only want the first or some two or three strings present in          the text file, but the whole of it, and we have no idea of what the length of the file is, we’ll use a while loop.
4. We’ll run the while loop until the file reaches the end of it, and that gets checked by using eof() , which returns 1 or true if the file reaches the end. Till then a 0 or false.
5. We’ll use getline to store the whole line in the string variable st. Don’t forget to include the header file <string>.
6. This program now successfully prints the whole content of the text file.

### C++ Templates: Must for Competitive Programming :

It has been quite a journey till here, and I feel grateful to have you all with me in the same. We have covered a lot in C++ and there is yet a great deal left. But we’ll make everything ahead a cakewalk together.

Today we have in the box, the most important topic for all you enthusiastic programmers, C++ templates. We’ll follow the below-mentioned roadmap:

1. What is a template in C++ programming?
2. Why templates?
3. Syntax

**What is a template in C++ programming?**

 A template is believed to escalate the potential of C++ several fold by giving it the ability to define data types as parameters making it useful to reduce repetitions of the same declaration of classes for different data types. Declaring classes for every other data type(which if counted is way too much) in the very first place violates the DRY( Don’t Repeat Yourself) rule of programming and on the other doesn't completely utilise the potential of C++.

It is very analogous to when we said classes are the templates for objects, here templates itself are the templates of the classes. That is, what classes are for objects, templates are for classes.

**Why templates?**

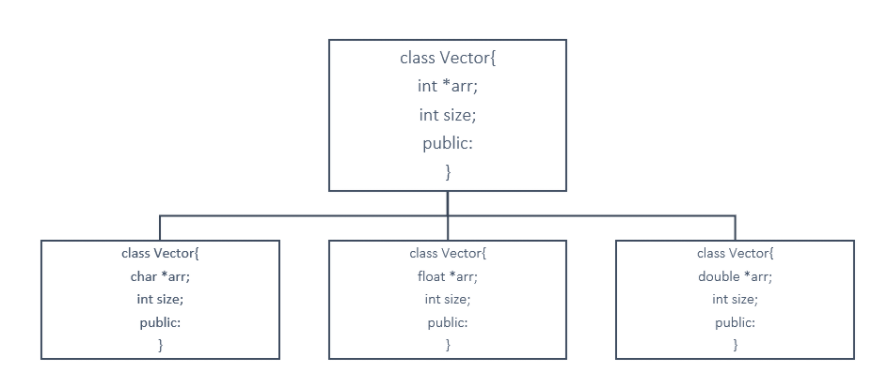
1. **DRY Rule:**

To understand the reason behind using templates, we will have to understand the effort behind declaring classes for different data types. Suppose we want to have a vector for each of the three(can be more) data types, int, float and char. Then we’ll obviously write the whole thing again and again making it awfully difficult. This is where the saviour comes, the templates. It helps parametrizing the data type and declaring it once in the source code suffice. Very similar to what we do in functions. It is because of this, also called, ‘parameterized classes’.

1. **Generic Programming:**

It is called generic, because it is sufficient to declare a template once, it becomes general and it works all along for all the data types.

Refer to the schematic below:



We had to copy the same thing again and again for different data types, but a template solves it all. Refer to the syntax section for how.

Below is the template for a vector of int data type, and it goes similarly for float char double, etc.

class vector {

int \*arr;

int size;

public:

};

**Syntax:**

Understanding the syntax below:

1. First, we declare a template of class and pass a variable T as its parameter.
2. Define the class of vector and keep the data type of \*arr as T only. Now, the array becomes of the type we supply in the template.

Now we can easily use this template to declare umpteen number of classes in our main scope. Be it int, float, or arr vector.

#include <iostream>

using namespace std;

template <class T>

class vector {

T \*arr;

int size;

public:

vector(T\* arr)[

//code

]

//and many other methods

};

int main() {

vector<int> myVec1();

vector<float> myVec2();

return 0;

}

Templates are believed to be very useful for people who pursue competitive programming. It makes their work several folds easier. It gives them an edge over others. It is a must because it saves you a lot of time while programming. And I believe you ain’t want to miss this opportunity to learn, right?

So, get to the playlist as soon as you can. Save yourselves some time and get over your competitors.

Thank you, friends, for being with me throughout, hope you liked the tutorial. And If you haven’t checked out the whole playlist yet, it’s never too late, move on to [codewithharry.com](https://www.codewithharry.com/) or my YouTube channel to access it. I hope you enjoy them. Templates are an inevitable part of this process of learning C++. You just cannot afford to miss this. In the next tutorial, we’ll be writing a program using templates for your better understanding, see you there, till then keep coding.

#include <iostream>

using namespace std;

template <class T>

class vector

{

public:

T \*arr;

int size;

vector(int m)

{

size = m;

arr = new T[size];

}

T dotProduct(vector &v){

T d=0;

for (int i = 0; i < size; i++)

{

d+=this->arr[i]\*v.arr[i];

}

return d;

}

};

int main()

{

vector<float> v1(3); //vector 1 with a float data type

v1.arr[0] = 1.4;

v1.arr[1] = 3.3;

v1.arr[2] = 0.1;

vector<float> v2(3); //vector 2 with a float data type

v2.arr[0]=0.1;

v2.arr[1]=1.90;

v2.arr[2]=4.1;

float a = v1.dotProduct(v2);

cout<<a<<endl;

return 0;

}

**Template With Multiple Parameters:**

template<class T1, class T2>

class myClass{

public:

T1 data1;

T2 data2;

myClass(T1 a,T2 b){

data1 = a;

data2 = b;

}

void display(){

cout<<this->data1<<" "<<this->data2;

}

};

int main()

{

myClass<int, char> obj(1, 'c');

obj.display();

}

### C++ Function Templates & Function Templates with Parameters:

template<class T1, class T2>

float funcAverage(T1 a, T2 b){

float avg= (a+b)/2.0;

return avg;

}

int main(){

float a;

a = funcAverage(5,2);

printf("The average of these numbers is %f",a);

return 0;

}

Copy

The average of these numbers is 3.500000

PS D:\MyData\Business\code playground\C++ course>

**Member Function Templates & Overloading Template Functions in C++ :**

template <class T>

class Harry

{

public:

T data;

Harry(T a)

{

data = a;

}

void display();

};

template <class T>

void Harry<T> :: display(){

cout<<data;

}

int main()

{

Harry<int> h(5.7);

cout << h.data << endl;

h.display();

return 0;

}

And the output is:

5

5

PS D:\MyData\Business\code playground\C++ course>

Copy

Now, we’ll move to the**overloading of a function template**. Overloading a function simply means assigning two or more functions with the same name, the same job, but with different parameters.  For that, we’ll declare a void function named func. And a template function with the same name. Follow the snippet below to do the same:

1. We made two void functions, one specified and one generic using a template.
2. The first one receives an integer and prints the integer with a different prefix.
3. The generic one receives the value as well as the data type and prints the value with a different prefix.
4. Now, we’ll wish to see the output of the following functions, by calling them from the main. Refer to the main program below the snippet below.

#include <iostream>

using namespace std;

void func(int a){

cout<<"I am first func() "<<a<<endl;

}

template<class T>

void func(T a){

cout<<"I am templatised func() "<<a<<endl;

}

int main()

{

func(4); //Exact match takes the highest priority

return 0;

}

And the output is,

I am first func() 4

PS D:\MyData\Business\code playground\C++ course>

### Standard Template Library (STL):

We have been waiting so long to start this, but creating a base is as important as any other phase. So, today we’ll be starting the most awaited topic, the STL( Standard Template Library).

There is a reason why I’ve been saying that this topic is a must for all the competitive programmers out there,so let’s deal with that first.

**Why is this important for competitive programmers?**

1. Competitive programming is a part of various environments, be it job interviews, coding contests and all, and if you’re in one of those environments, you’ll be given limited time to code your program.
2. So, suppose you want in your program, a resizable array, or sort an array or any other data structure. or search for some element in your container.
3. You will always try to code a function which will execute the above mentioned things, and end up losing a great amount of time. But here is when you will use STL.

An STL is a library of generic functions and classes which saves you time and energy which you would have spent constructing for your use. This helps you reuse these well tested classes and functions umpteenth number of times according to your own convenience.

To put this simply, STL is used because it is not a good idea to reinvent something which is already built and can be used to innovate things further. Suppose you go to a company who builds cars, they will not ask you to start from scratch, but to start from where it is left. This is the basic idea behind using STL.

**COMPONENTS OF STL:**

We have three components in STL:

1. Containers
2. Algorithm
3. Iterators

Let’s deal with them individually;

**Containers:**

Container is an object which stores data. We have different containers having their own benefits. These are the implemented template classes for our use, which can be used just by including this library. You can even customise these template classes.

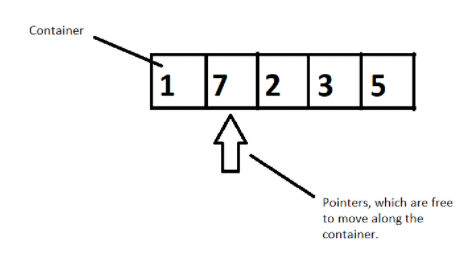
**Algorithms:**

Algorithms are a set of instructions which manipulates the input data to arrive at some desired result. In STL, we have already written algorithms, for example, to sort some data structure, or search some element in an array. These algorithms use template functions.

**Iterators:**

Iterators are objects which refer to an element in a container. And we handle them very much similarly to a pointer. Their basic job is to connect algorithms to the container and play a vital role in manipulation of the data.

I’ll give you a quick illustration of how they work combinedly.



**Figure 1: Illustration of how these three components work together**

Suppose we have a container of integers, and we want to sort them in ascending order. We will have pointers which will help moving elements to places by pointing to it, following a well-constructed algorithm. So, here a container gets sorted by following an algorithm by the use of pointers. This is how they work in accordance with each other.

So, this was the basics of STL and the motivation behind using it in your programs. I hope I was able to introduce it to you.

Thank you, for being with me throughout, hope you liked the tutorial. If you haven’t checked out the whole playlist yet, move on to [codewithharry.com](https://www.codewithharry.com/) or my YouTube channel to access it. I hope you enjoy them all. See you all in the next tutorial where we’ll dive deep in the containers and its different types. Till then keep coding.

### Containers in C++ STL:

**Containers**, objects which store data, **Algorithms,**set of procedures to process data, and **Iterators,**objects which point to some element in a container. Today, in this tutorial, we will be interested in discussing more about containers**.**

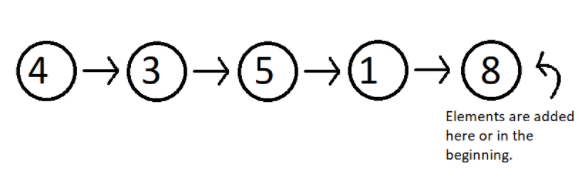
Containers are themselves of three types:

1. Sequence Containers
2. Associative Containers
3. Derived Containers

When we talked about containers, we said containers are objects which store data, but what are its three types all about? We’ll discuss that too.

* **Sequence Containers:**

A **sequence container**stores that data in a linear fashion. Refer to the illustration below to understand what storing something in a linear fashion means.

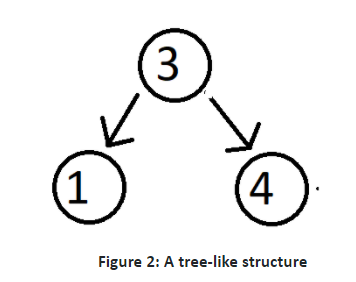


**Figure 1: Elements stored in a linear fashion**

Sequence containers include **Vector, List, Dequeue etc.**These are some of the most used sequence containers.

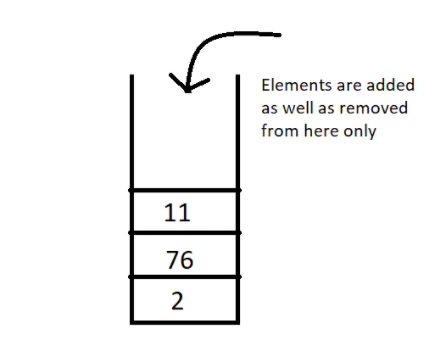
* **Associative Containers:**

An **associative container**is designed in such a way that enhances the accessing of some element in that container. It is very much used when the user wants to fastly reach some element. Some of these containers are, **Set, Multiset, Map, Multimap etc.**They store their data in a tree-like structure.



* **Derived Containers :**

As the name suggests, these containers are derived from either the sequence or the associative containers. They often provide you with some better methods to deal with your data. They deal with real life modelling. Some examples of derived containers are **Stack, Queue, Priority Queue, etc.**The following illustration give you the idea of how a stack works.



**Figure 3: A stack, works on the first in first out [FIFO] method**

Now since we have got the basic idea of all the three types of containers, a question which might arise is **when to use which**.  So, let’s deal with that,

In sequence containers, we have **Vectors**, which has following properties:

1. Faster random access to elements in comparison to array
2. Slower insertion and deletion at some random position, except at the end.
3. Faster insertion at the end.

In **Lists,**we have,

1. Random accessing elements is too slow, because every element is traversed using pointers.
2. Insertion and deletion at any position is relatively faster, because they only use pointers, which can easily be manipulated.

In associative containers, every operation except random access is faster in comparison to any other containers, be it inserting or deleting any element.

In associative containers, we cannot specifically tell which operation is faster or slower, we’ll have to inspect every data structure separately, and to get a clearer picture of all of these, you can access my Data Structure course : [Data Structures and Algorithms Course in Hindi](https://youtube.com/playlist?list=PLu0W_9lII9ahIappRPN0MCAgtOu3lQjQi)

For now, I'd like to hold on to our topic STL, and get you a strong hold on this too. In the coming videos, we’ll deal with our vectors, list, dequeues, set, multiset, maps, stack and much more. Just bear with me.

Thank you, for being with me throughout, hope you liked the tutorial. If you haven’t checked out the whole playlist yet, move on to [codewithharry.com](https://www.codewithharry.com/) or my YouTube channel to access it. I hope you enjoy them all. See you all in the next tutorial where we’ll talk about Vectors in C++ STL in detail. Till then keep coding.

**Vector In STL:**

Vectors are sequence containers representing arrays that can change in size.  
  
Just like arrays, vectors use contiguous storage locations for their elements, which means that their elements can also be accessed using offsets on regular pointers to its elements, and just as efficiently as in arrays. But unlike arrays, their size can change dynamically, with their storage being handled automatically by the container.  
  
Internally, vectors use a dynamically allocated array to store their elements. This array may need to be reallocated in order to grow in size when new elements are inserted, which implies allocating a new array and moving all elements to it. This is a relatively expensive task in terms of processing time, and thus, vectors do not reallocate each time an element is added to the container.  
  
Instead, vector containers may allocate some extra storage to accommodate for possible growth, and thus the container may have an actual [capacity](https://www.cplusplus.com/vector::capacity) greater than the storage strictly needed to contain its elements (i.e., its [size](https://www.cplusplus.com/vector::size)). Libraries can implement different strategies for growth to balance between memory usage and reallocations, but in any case, reallocations should only happen at logarithmically growing intervals of [size](https://www.cplusplus.com/vector::size) so that the insertion of individual elements at the end of the vector can be provided with *amortized constant time* complexity (see [push\_back](https://www.cplusplus.com/vector::push_back)).  
  
Therefore, compared to arrays, vectors consume more memory in exchange for the ability to manage storage and grow dynamically in an efficient way.  
  
Compared to the other dynamic sequence containers ([deques](https://www.cplusplus.com/deque), [lists](https://www.cplusplus.com/list) and [forward\_lists](https://www.cplusplus.com/forward_list)), vectors are very efficient accessing its elements (just like arrays) and relatively efficient adding or removing elements from its [end](https://www.cplusplus.com/vector::end). For operations that involve inserting or removing elements at positions other than the end, they perform worse than the others, and have less consistent iterators and references than [lists](https://www.cplusplus.com/list) and [forward\_lists](https://www.cplusplus.com/forward_list).

### Member functions

[**(constructor)**](https://www.cplusplus.com/reference/vector/vector/vector/)

Construct vector (public member function )

[**(destructor)**](https://www.cplusplus.com/reference/vector/vector/~vector/)

Vector destructor (public member function )

[**operator=**](https://www.cplusplus.com/reference/vector/vector/operator=/)

Assign content (public member function )

**Iterators**:

[**begin**](https://www.cplusplus.com/reference/vector/vector/begin/)

Return iterator to beginning (public member function )

[**end**](https://www.cplusplus.com/reference/vector/vector/end/)

Return iterator to end (public member function )

[**rbegin**](https://www.cplusplus.com/reference/vector/vector/rbegin/)

Return reverse iterator to reverse beginning (public member function )

[**rend**](https://www.cplusplus.com/reference/vector/vector/rend/)

Return reverse iterator to reverse end (public member function )

[**cbegin**](https://www.cplusplus.com/reference/vector/vector/cbegin/)

Return const\_iterator to beginning (public member function )

[**cend**](https://www.cplusplus.com/reference/vector/vector/cend/)

Return const\_iterator to end (public member function )

[**crbegin**](https://www.cplusplus.com/reference/vector/vector/crbegin/)

Return const\_reverse\_iterator to reverse beginning (public member function )

[**crend**](https://www.cplusplus.com/reference/vector/vector/crend/)

Return const\_reverse\_iterator to reverse end (public member function )

**Capacity**:

[**size**](https://www.cplusplus.com/reference/vector/vector/size/)

Return size (public member function )

[**max\_size**](https://www.cplusplus.com/reference/vector/vector/max_size/)

Return maximum size (public member function )

[**resize**](https://www.cplusplus.com/reference/vector/vector/resize/)

Change size (public member function )

[**capacity**](https://www.cplusplus.com/reference/vector/vector/capacity/)

Return size of allocated storage capacity (public member function )

[**empty**](https://www.cplusplus.com/reference/vector/vector/empty/)

Test whether vector is empty (public member function )

[**reserve**](https://www.cplusplus.com/reference/vector/vector/reserve/)

Request a change in capacity (public member function )

[**shrink\_to\_fit**](https://www.cplusplus.com/reference/vector/vector/shrink_to_fit/)

Shrink to fit (public member function )

**Element access**:

[**operator[]**](https://www.cplusplus.com/reference/vector/vector/operator%5b%5d/)

Access element (public member function )

[**at**](https://www.cplusplus.com/reference/vector/vector/at/)

Access element (public member function )

[**front**](https://www.cplusplus.com/reference/vector/vector/front/)

Access first element (public member function )

[**back**](https://www.cplusplus.com/reference/vector/vector/back/)

Access last element (public member function )

[**data**](https://www.cplusplus.com/reference/vector/vector/data/)

Access data (public member function )

**Modifiers**:

[**assign**](https://www.cplusplus.com/reference/vector/vector/assign/)

Assign vector content (public member function )

[**push\_back**](https://www.cplusplus.com/reference/vector/vector/push_back/)

Add element at the end (public member function )

[**pop\_back**](https://www.cplusplus.com/reference/vector/vector/pop_back/)

Delete last element (public member function )

[**insert**](https://www.cplusplus.com/reference/vector/vector/insert/)

Insert elements (public member function )

[**erase**](https://www.cplusplus.com/reference/vector/vector/erase/)

Erase elements (public member function )

[**swap**](https://www.cplusplus.com/reference/vector/vector/swap/)

Swap content (public member function )

[**clear**](https://www.cplusplus.com/reference/vector/vector/clear/)

Clear content (public member function )

[**emplace**](https://www.cplusplus.com/reference/vector/vector/emplace/)

Construct and insert element (public member function )

[**emplace\_back**](https://www.cplusplus.com/reference/vector/vector/emplace_back/)

Construct and insert element at the end (public member function )

**Allocator**:

[**get\_allocator**](https://www.cplusplus.com/reference/vector/vector/get_allocator/)

Get allocator (public member function )