Contents

[Contents 1](#_Toc515622501)

[1 Interview Questions and Answers for in C++ 2](#_Toc515622502)

[1.1 What is the Order of Constructor/ Destructor Call in C++ ? 2](#_Toc515622503)

[1.1.1 Example: 2](#_Toc515622504)

[1.2 How to call the parameterized constructor of base class in derived class constructor? 3](#_Toc515622505)

[1.3 What is Static Keyword in C++? 4](#_Toc515622506)

[1.3.1 Static variables in a Function: 4](#_Toc515622507)

[1.3.2 Static variables in a Class: 4](#_Toc515622508)

[1.3.3 Static Members of Class 5](#_Toc515622509)

[1.4 What is Memory Leak? How can we avoid? 7](#_Toc515622510)

[1.5 What are Dangling, Void , Null and Wild Pointers ? 7](#_Toc515622511)

[1.5.1 Dangling pointer 7](#_Toc515622512)

[1.5.2 Void pointer 7](#_Toc515622513)

[1.5.3 NULL Pointer 8](#_Toc515622514)

[1.5.4 Wild pointer 8](#_Toc515622515)

[1.6 How to sort an array of dates in C/C++? 9](#_Toc515622516)

# Interview Questions and Answers for in C++

## What is the Order of Constructor/ Destructor Call in C++ ?

Whenever we create an object of a class, the default constructor of that class is invoked automatically to initialize the members of the class.

If we inherit a class from another class and create an object of the derived class, it is clear that the default constructor of the derived class will be invoked but before that the default constructor of all of the base classes will be invoke, i.e the order of invokation is that the base class’s default constructor will be invoked first and then the derived class’s default constructor will be invoked.

### Example:

// C++ program to show the order of constructor calls

// in Multiple Inheritance

#include <iostream>

using namespace std;

class Parent1 // first base class

{

public:

Parent1() // first base class's Constructor

{

cout << "Inside first base class" << endl;

}

};

class Parent2 // second base class

{

public:

Parent2() // second base class's Constructor

{

cout << "Inside second base class" << endl;

}

};

class Child : public Parent1, public Parent2 // child class inherits Parent1 and Parent2

{

public:

Child() // child class's Constructor

{

cout << "Inside child class" << endl;

}

};

int main() {

Child obj1; // child class's Constructor

return 0;

}

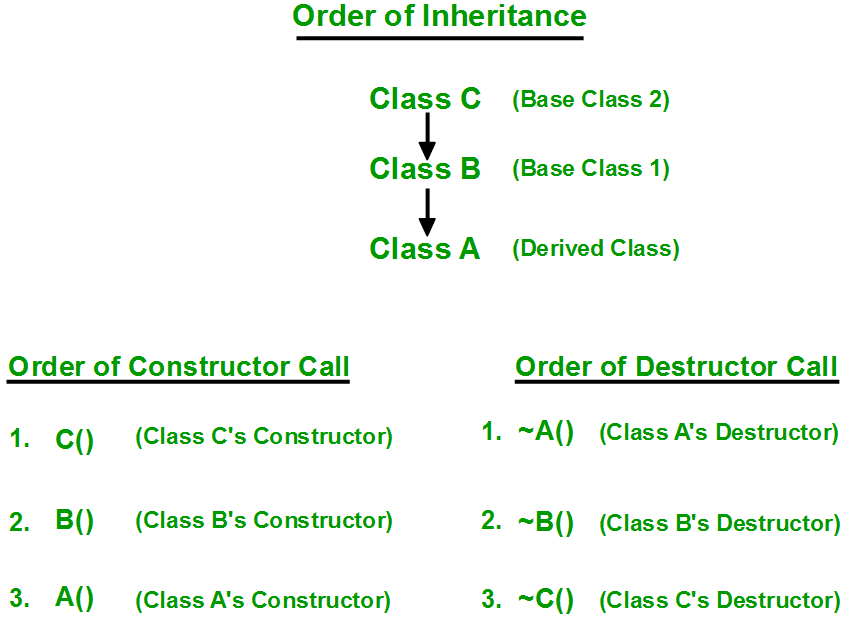
Output:

Inside first base class

Inside second base class

Inside child class

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



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

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

// C++ program to show how to call parameterised Constructor

// of base class when derived class's Constructor is called

#include <iostream>

using namespace std;

class Parent // base class

{

public:

Parent(int i) // base class's parameterised constructor

{

int x = i;

cout << "Inside base class's parameterised constructor" << endl;

}

};

class Child : public Parent // sub class

{

public:

Child(int j) : Parent(j) // sub class's parameterised constructor

{

cout << "Inside sub class's parameterised constructor" << endl;

}

};

int main() {

Child obj1(10); // creating object of class Child

return 0;

}

Output:

Inside base class's parameterized constructor

Inside sub class's parameterized constructor

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

## What is Static Keyword in C++?

<https://www.geeksforgeeks.org/static-keyword-cpp/>

Static keyword has different meanings when used with different types. We can use static keyword with:

**Static Variables** : Variables in a function, Variables in a class

**Static Members of Class** : Class objects and Functions in a class

Let us now look at each one of these use of static in details:

### Static variables in a Function:

When a variable is declared as static, space for **it gets allocated for the lifetime of the program**. Even if the function is called multiple times, space for the static variable is allocated only once and the value of variable in the previous call gets carried through the next function call. This is useful for implementing co-routines in C/C++ or any other application where previous state of function needs to be stored.

// C++ program to demonstrate the use of static Static variables in a Function

#include <iostream>

#include <string>

using namespace std;

void demo()

{

static int count = 0; // static variable

cout << count << " ";

count++; // value is updated and will be carried to next function calls

}

int main()

{

for (int i = 0; i<5; i++)

demo();

return 0;

}

Output:

0 1 2 3 4

You can see in the above program that the variable count is declared as static. So, its value is carried through the function calls. The variable count is not getting initialized for every time the function is called.

### Static variables in a Class:

As the variables declared as static are initialized only once as they are allocated space in separate static storage so, the static variables **in a class are not shared by the objects**. There cannot be multiple copies of same static variables for different objects. Also because of this reason static variables cannot be initialized using constructors.

// C++ program to demonstrate static variables inside a class

#include<iostream>

using namespace std;

class GfG {

public:

static int i;

GfG()

{ };// Do nothing

};

**int GfG::i = 1;** // initialize the static variable

int main()

{

GfG obj;

cout << obj.i; // prints value of i

}

Output:

**1**

### Static Members of Class

***Class objects as static:***

Just like variables, objects also when declared as static have a scope till the lifetime of program.Consider the below program where the **object is non-static**.

// CPP program to illustrate when not using static keyword

#include<iostream>

using namespace std;

class GfG

{

int i;

public:

GfG()

{

i = 0;

cout << "Inside Constructor\n";

}

~GfG()

{

cout << "Inside Destructor\n";

}

};

int main()

{

int x = 0;

if (x == 0)

{

GfG obj;

}

cout << "End of main\n";

}

Output:

Inside Constructor

End of main

Inside Destructor

In the above program the object is declared inside if block as non-static. So, the scope of variable is inside the if block only. So when the object is created the constructor is invoked and soon as the control of if block gets over the destructor is invoked as the scope of object is inside the if block only where it is declared.  
Let us now see the change in output if we declare the object as static.

// CPP program to illustrate

// class objects as static

#include<iostream>

using namespace std;

class GfG {

int i = 0;

public:

GfG()

{

i = 0;

cout << "Inside Constructor\n";

}

~GfG()

{

cout << "Inside Destructor\n";

}

};

int main(){

int x = 0;

if (x == 0)

{

static GfG obj;

}

cout << "End of main\n";

}

OUTPUT:

Inside Constructor

Inside Destructor

End of main

In the above program the object is declared inside the if block as non-static. So, the scope of variable is inside the if block only. So when the object is created the constructor is invoked and soon as the control of if block gets over the destructor is invoked as the scope of object is inside the if block only where it is declared.

Let us now see the change in output if we declare the object as static.

// CPP program to illustrate

// class objects as static

#include<iostream>

using namespace std;

class GfG

{

int i = 0;

public:

GfG()

{

i = 0;

cout << "Inside Constructor\n";

}

~GfG()

{

cout << "Inside Destructor\n";

}

};

int main()

{

int x = 0;

if (x == 0)

{

static GfG obj;

}

cout << "End of main\n";

}

OUTPUT:

Inside Constructor

End of main

Inside Destructor

You can clearly see the change in output. Now the destructor is invoked after the end of main. This happened because the scope of static object is throughout the life time of program.

***Static functions in a class:***

Just like the static data members or static variables inside the class, static member functions also does not depend on object of class. We are allowed to invoke a static member function using the object and the ‘.’ operator but it is recommended to invoke the static members using the class name and the scope resolution operator.

**Static member functions are allowed to access only the static data members or other static member functions**, they cannot access the non-static data members or member functions of the class.

// C++ program to demonstrate static member function in a class

#include<iostream>

using namespace std;

class GfG

{

public:

static void printMsg() // static member function

{

cout << "Welcome to GfG!";

}

};

int main()

{

GfG::printMsg(); // invoking a static member function

}

OUTPUT:

Welcome to GfG!

## What is Memory Leak? How can we avoid?

Memory leak occurs when programmers create a memory in heap and forget to delete it.

Memory leaks are particularly serious issues for programs like daemons and servers which by definition never terminates.

/\* Function with memory leak \*/

#include <stdlib.h>

void f()

{

int \*ptr = (int \*)malloc(sizeof(int));

/\* Do some work \*/

return; /\* Return without freeing ptr\*/

}

//To avoid memory leaks, memory allocated on heap should always be freed when no longer needed.

void f()

{

int \*ptr = (int \*)malloc(sizeof(int));

/\* Do some work \*/

free(ptr);

return;

}

## What are Dangling, Void , Null and Wild Pointers ?

### Dangling pointer

A pointer pointing to a memory location that has been deleted (or freed) is called dangling pointer.

// Deallocating a memory pointed by ptr causes

// dangling pointer

#include <stdlib.h>

#include <stdio.h>

int main()

{

int \*ptr = (int \*)malloc(sizeof(int));

// After below free call, ptr becomes a

// dangling pointer

free(ptr);

// No more a dangling pointer

ptr = NULL;

}

### [Void pointer](http://quiz.geeksforgeeks.org/void-pointer-c/)

Void pointer is a specific pointer type – void \* – a pointer that points to some data location in storage, which doesn’t have any specific type. Void refers to the type. Basically the type of data that it points to is can be any. If we assign address of char data type to void pointer it will become char Pointer, if int data type then int pointer and so on. Any pointer type is convertible to a void pointer hence it can point to any value.

**Important Points:**

* void pointers cannot be dereferenced. It can however be done using typecasting the void pointer
* Pointer arithmetic is not possible on pointers of void due to lack of concrete value and thus size.

#include<stdlib.h>

int main()

{

int x = 4;

float y = 5.5;

//A void pointer

void \*ptr;

ptr = &x;

// (int\*)ptr - does type casting of void

// \*((int\*)ptr) dereferences the typecasted

// void pointer variable.

printf("Integer variable is = %d", \*((int\*)ptr));

// void pointer is now float

ptr = &y;

printf("\nFloat variable is= %f", \*((float\*)ptr));

return 0;

}

Output:

Integer variable is = 4

Float variable is= 5.500000

### NULL Pointer

NULL Pointer is a pointer which is pointing to nothing. In case, if we don’t have address to be assigned to a pointer, then we can simply use NULL.

#include <stdio.h>

int main()

{

// Null Pointer

int \*ptr = NULL;

printf("The value of ptr is %u", ptr);

return 0;

}

*Output:*

The value of ptr is 0

**Important Points**

NULL vs Uninitialized pointer – An uninitialized pointer stores an undefined value. A null pointer stores a defined value, but one that is defined by the environment to not be a valid address for any member or object.

NULL vs Void Pointer – Null pointer is a value, while void pointer is a type

### [Wild pointer](https://www.geeksforgeeks.org/what-are-wild-pointers-how-can-we-avoid/)

A pointer which has not been initialized to anything (not even NULL) is known as wild pointer. The pointer may be initialized to a non-NULL garbage value that may not be a valid address.

int main(){

int \*p; /\* wild pointer \*/

int x = 10;

// p is not a wild pointer now

p = &x;

return 0;

}

## How to sort an array of dates in C/C++?

// C++ program to sort an array of dates

#include <iostream>

#include <algorithm>

using namespace std;

// Structure for date

struct Date{

int day, month, year;

};

// This is the compare function used by in-built sort

// function to sort the array of dates.

// It takes two Dates as parameters (const is

// given to tell the compiler that the value won't be

// changed during the compare - this is for optimisation..)

// Returns true if dates have to be swapped and returns

// false if not. Since we want ascending order, we return

// true if first Date is less than second date

bool compare(const Date &d1, const Date &d2){

// All cases when true should be returned

if (d1.year < d2.year)

return true;

if (d1.year == d2.year && d1.month < d2.month)

return true;

if (d1.year == d2.year && d1.month == d2.month &&

d1.day < d2.day)

return true;

return false; // If none of the above cases satisfy, return false

}

// Function to sort array arr[0..n-1] of dates

void sortDates(Date arr[], int n){

// Calling in-built sort function.

// First parameter array beginning,

// Second paramter - array ending,

// Third is the custom compare function

sort(arr, arr + n, compare);

}

// Driver Program

int main(){

Date arr[] = { { 20, 1, 2014 },

{ 25, 3, 2010 },

{ 3, 12, 1676 },

{ 18, 11, 1982 },

{ 19, 4, 2015 },

{ 9, 7, 2015 } };

int n = sizeof(arr) / sizeof(arr[0]);

sortDates(arr, n);

cout << "Sorted dates are\n";

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

{

cout << arr[i].day << " " << arr[i].month

<< " " << arr[i].year;

cout << endl;

}

}

## What is the declaration of “NULL”?

#define NULL ((char \*)0)

or

#define NULL 0L

or

#define NULL 0