Constructors in C++

Difficulty Level : Easy • Last Updated : 27 Mar, 2023

Constructor in C++ is a special method that is invoked automatically at the time of object creation. It is used to initialize the data members of new objects generally. The constructor in C++ has the same name as the class or structure. Constructor is invoked at the time of object creation. It constructs the values i.e. provides data for the object which is why it is known as constructors.

• Constructor is a member function of a class, whose name is same as the class name.
• Constructor is a special type of member function that is used to initialize the data members for an object of a class automatically, when an object of the same class is created.
• Constructor is invoked at the time of object creation. It constructs the values i.e. provides data for the object that is why it is known as constructor.
• Constructor do not return value, hence they do not have a return type.

```
The prototype of the constructor looks like
    <class-name> (list-of-parameters);



Constructor can be defined inside the class declaration or outside the class
declaration

a.     Syntax for defining the constructor within the class

        <class-name>(list-of-parameters)
        {
                  //constructor definition
    }
```

b.   Syntax for defining the constructor outside the class

```
    <class-name>: :<class-name>(list-of-parameters)
    {
            //constructor definition
    }
```

AD

C++

```
// Example: defining the constructor within the class

#include<iostream>
using namespace std;
class student
{
    int rno;
    char name[50];
    double fee;
    public:
    student()
    {
        cout<<"Enter the RollNo:";
        cin>>rno;
        cout<<"Enter the Name:";
        cin>>name;
        cout<<"Enter the Fee:";
        cin>>fee;
    }


    void display()
    {
        cout<<endl<<rno<<"\t"<<name<<"\t"<<fee;
    }
};
```

```
int main()
{
    student s;  //constructor gets called automatically when we create the object of t
    s.display();
    return 0;

}
```

C++

```
// Example: defining the constructor outside the class

#include<iostream>
using namespace std;
class student
{
    int rno;
    char name[50];
    double fee;
    public:
    student();
    void display();

};

    student::student()
    {
        cout<<"Enter the RollNo:";
        cin>>rno;
        cout<<"Enter the Name:";
        cin>>name;
        cout<<"Enter the Fee:";
        cin>>fee;
    }

   void student::display()
    {
        cout<<endl<<rno<<"\t"<<name<<"\t"<<fee;
    }

int main()
{
    student s;
    s.display();
    return 0;
}
```

Characteristics of constructor

- The name of the constructor is same as its class name.
- Constructors are mostly declared in the public section of the class though it can be declared in the private section of the class.
- Constructors do not return values; hence they do not have a return type.
- A constructor gets called automatically when we create the object of the class.
- Constructors can be overloaded.
- Constructor can not be declared virtual.

Types of constructor

- Default constructor
- Parameterized constructor
- Overloaded constructor
- Constructor with default value
- Copy constructor
- Inline constructor

Constructor does not have a return value, hence they do not have a return type.

The prototype of Constructors is as follows:

```
<class-name> (list-of-parameters);
```

Constructors can be defined inside or outside the class declaration:-

The syntax for defining the constructor within the class:

```
<class-name> (list-of-parameters) { // constructor definition }
```

The syntax for defining the constructor outside the class:

```
<class-name>: :<class-name> (list-of-parameters){ // constructor definition}
```

Example

C++

```
// defining the constructor within the class

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

```
class student {
    int rno;
    char name[10];
    double fee;

public:
    student()
    {
        cout << "Enter the RollNo:";
        cin >> rno;
        cout << "Enter the Name:";
        cin >> name;
        cout << "Enter the Fee:";
        cin >> fee;
    }

    void display()
    {
        cout << endl << rno << "\t" << name << "\t" << fee;
    }
};

int main()
{
    student s; // constructor gets called automatically when
               // we create the object of the class
    s.display();

    return 0;
}
```

Output

```
Enter the RollNo:Enter the Name:Enter the Fee:
0         6.95303e-310
```

Example

C++

```
// defining the constructor outside the class

#include <iostream>
using namespace std;
class student {
    int rno;
    char name[50];
    double fee;

public:
    student();
    void display();
};

student::student()
{
    cout << "Enter the RollNo:";
    cin >> rno;

    cout << "Enter the Name:";
    cin >> name;

    cout << "Enter the Fee:";
    cin >> fee;
}

void student::display()
{
    cout << endl << rno << "\t" << name << "\t" << fee;
}

int main()
{
    student s;
    s.display();

    return 0;
}
```

Output:

```
Enter the RollNo: 30
Enter the Name: ram
Enter the Fee: 20000
30 ram 20000
```

How constructors are different from a normal member function?

C++

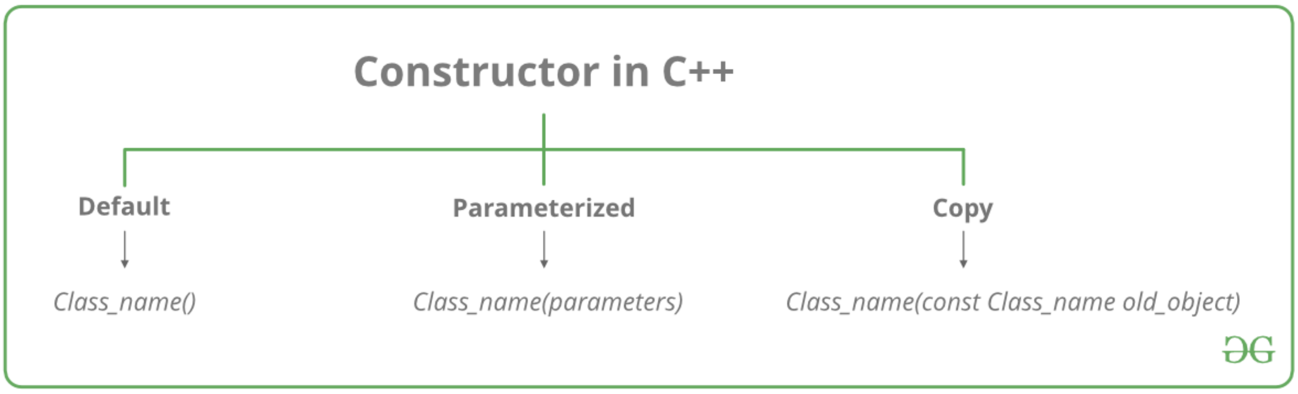
```
#include <iostream>
using namespace std;

class Line {
  public:
     void setLength( double len );
     double getLength( void );
     Line( double len ); //This is the constructor
  private:
     double length;
};
//Member function definition including constructor
Line::Line( double len ) {
  cout<<"Object is being created , length ="<< len <<endl;
  length = len;
}
void Line::setLength( double len ) {
  length = len;
}
double Line::getLength( void ) {
  return length;
}
//Main function for the program
int main() {
  Line line(10.0);
  //get initially set length
  cout<<"Length of line :" << line.getLength() << endl;
  //set line length again
  line.setLength(6.0);
  cout<<"Length of line :" << line.getLength() << endl;

  return 0;
}
```

A constructor is different from normal functions in following ways:

- Constructor has same name as the class itself
- Default Constructors don't have input argument however, Copy and Parameterized Constructors have input arguments
- 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).



Let us understand the types of constructors in C++ by taking a real-world example. Suppose you went to a shop to buy a marker. When you want to buy a marker, what are the options. The first one you go to a shop and say give me a marker. So just saying give me a marker mean that you did not set which brand name and which color, you didn't mention anything just say you want a marker. So when we said just I want a marker so whatever the frequently sold marker is there in the market or in his shop he will simply hand over that. And this is what a default constructor is! The second method is you go to a shop and say I want a marker a red in color and XYZ brand. So you are mentioning this and he will give you that marker. So in this case you have given the parameters. And this is what a parameterized constructor is! Then the third one you go to a shop and say I want a marker like this(a physical marker on your hand). So the shopkeeper will see that marker. Okay, and he will give a new marker for you. So copy of that marker. And that's what a copy constructor is!

Characteristics of the constructor:

- The name of the constructor is the same as its class name.
- Constructors are mostly declared in the public section of the class though it can be declared in the private section of the class.
- Constructors do not return values; hence they do not have a return type.
- A constructor gets called automatically when we create the object of the class.
- Constructors can be overloaded.
- Constructor can not be declared virtual.
- Constructor cannot be inherited.
- Addresses of Constructor cannot be referred.
- Constructor make implicit calls to **new** and **delete** operators during memory allocation.

Types of Constructors

1. Default Constructors: Default constructor is the constructor which doesn't take any argument. It has no parameters. It is also called a zero-argument constructor.

CPP

```
// Cpp program to illustrate the
// concept of Constructors
#include <iostream>
using namespace std;

class construct {
public:
    int a, b;

    // Default Constructor
    construct()
    {
        a = 10;
        b = 20;
    }
};

int main()
{
    // Default constructor called automatically
    // when the object is created
    construct c;
    cout << "a: " << c.a << endl << "b: " << c.b;
    return 1;
}
```

Output

```
a: 10
b: 20
```

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

C++

```
// Example
#include<iostream>
using namespace std;
class student
```

```
{
    int rno;
    char name[50];
    double fee;
    public:
    student()                     //  Explicit Default constructor
    {
        cout<<"Enter the RollNo:";
        cin>>rno;
        cout<<"Enter the Name:";
        cin>>name;
        cout<<"Enter the Fee:";
        cin>>fee;
    }

    void display()
    {
        cout<<endl<<rno<<"\t"<<name<<"\t"<<fee;
    }
};

int main()
{
    student s;
    s.display();
    return 0;
}
```

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.

> ***Note:*** *when the parameterized constructor is defined and no default constructor is defined explicitly, the compiler will not implicitly call the default constructor and hence creating a simple object as*
>
> ```
> Student s;
> Will flash an error
> ```

CPP

```
// CPP program to illustrate
// parameterized constructors
#include <iostream>
using namespace std;
```

```
class Point {
private:
    int x, y;

public:
    // Parameterized Constructor
    Point(int x1, int y1)
    {
        x = x1;
        y = y1;
    }

    int getX() { return x; }
    int getY() { return y; }
};

int main()
{
    // Constructor called
    Point p1(10, 15);

    // Access values assigned by constructor
    cout << "p1.x = " << p1.getX()
         << ", p1.y = " << p1.getY();

    return 0;
}
```

Output

```
p1.x = 10, p1.y = 15
```

C++

```
// Example

#include<iostream>
#include<string.h>
using namespace std;

class student
{
    int rno;
    char name[50];
    double fee;

      public:
    student(int,char[],double);
    void display();

};
```

```
student::student(int no,char n[],double f)
{
    rno=no;
    strcpy(name,n);
    fee=f;
}

void student::display()
{
    cout<<endl<<rno<<"\t"<<name<<"\t"<<fee;
}

int main()
{
    student s(1001,"Ram",10000);
    s.display();
    return 0;
}
```

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.

3. Copy Constructor:

A copy constructor is a member function that initializes an object using another object of the same class. A detailed article on Copy Constructor.

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

Copy constructor takes a reference to an object of the same class as an argument.

```
Sample(Sample &t)
          {
                     id=t.id;
          }
```

CPP

```cpp
// Illustration
#include <iostream>
using namespace std;

class point {
private:
    double x, y;

public:
    // Non-default Constructor &
    // default Constructor
    point(double px, double py) { x = px, y = py; }
};

int main(void)
{

    // Define an array of size
    // 10 & of type point
    // This line will cause error
    point a[10];

    // Remove above line and program
    // will compile without error
    point b = point(5, 6);
}
```

Output:

```
Error: point (double px, double py): expects 2 arguments, 0 provided
```

C++

```cpp
// Implicit copy constructor

#include<iostream>
using namespace std;

class Sample
{          int id;
    public:
    void init(int x)
    {
        id=x;
    }
    void display()
    {
        cout<<endl<<"ID="<<id;
    }
};

int main()
{
    Sample obj1;
    obj1.init(10);
    obj1.display();

    Sample obj2(obj1); //or obj2=obj1;
    obj2.display();
    return 0;
}
```

Output

```
ID=10
ID=10
```

C++

```
// Example: Explicit copy constructor

#include <iostream>
using namespace std;

class Sample
{
    int id;
    public:
    void init(int x)
    {
        id=x;
    }
    Sample(){}  //default constructor with empty body

    Sample(Sample &t)   //copy constructor
    {
        id=t.id;
    }
    void display()
    {
        cout<<endl<<"ID="<<id;
    }
};
int main()
{
    Sample obj1;
    obj1.init(10);
    obj1.display();

    Sample obj2(obj1); //or obj2=obj1;    copy constructor called
    obj2.display();
    return 0;
}
```

Output

```
ID=10
ID=10
```

C++

```
#include<iostream>
#include<string.h>
using namespace std;
class student
{
    int rno;
    char name[50];
    double fee;
    public:
    student(int,char[],double);
    student(student &t)       //copy constructor
    {
        rno=t.rno;
        strcpy(name,t.name);
        fee=t.fee;
    }
    void display();

};




    student::student(int no,char n[],double f)
    {
        rno=no;
        strcpy(name,n);
        fee=f;
    }

   void student::display()
    {
        cout<<endl<<rno<<"\t"<<name<<"\t"<<fee;
    }

int main()
{
    student s(1001,"Manjeet",10000);
    s.display();

    student manjeet(s);   //copy constructor called
    manjeet.display();

    return 0;
}
```

C++

```
#include<iostream>
#include<string.h>
using namespace std;
class student
{
    int rno;
    char name[50];
    double fee;
    public:
    student(int,char[],double);
    student(student &t)       //copy constructor (member wise initialization)
    {
        rno=t.rno;
        strcpy(name,t.name);

    }
    void display();
    void disp()
    {
        cout<<endl<<rno<<"\t"<<name;
    }

};
    student::student(int no, char n[],double f)
    {
        rno=no;
        strcpy(name,n);
        fee=f;
    }

   void student::display()
    {
        cout<<endl<<rno<<"\t"<<name<<"\t"<<fee;
    }



int main()
{
    student s(1001,"Manjeet",10000);
    s.display();

    student manjeet(s);   //copy constructor called
    manjeet.disp();

    return 0;
}
```

Destructor:

A destructor is also a special member function as a constructor. Destructor destroys the class objects created by the constructor. Destructor has the same name as their class name preceded by a tilde (~) symbol. It is not possible to define more than one destructor. The destructor is only one way to destroy the object created by the constructor. Hence destructor can-not be overloaded. Destructor neither requires any argument nor returns any value. It is automatically called when the object goes out of scope. Destructors release memory space occupied by the objects created by the constructor. In destructor, objects are destroyed in the reverse of object creation.

The syntax for defining the destructor within the class

```
~ <class-name>()
{
  }
```

The syntax for defining the destructor outside the class

```
<class-name>: : ~ <class-name>(){}
```

C++

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

class Test {
public:
    Test() { cout << "\n Constructor executed"; }

    ~Test() { cout << "\n Destructor executed"; }
};
main()
{
    Test t;

    return 0;
}
```

Output

```
Constructor executed
Destructor executed
```

C++

```
#include <iostream>
using namespace std;
class Test {
public:
    Test() { cout << "\n Constructor executed"; }

    ~Test() { cout << "\n Destructor executed"; }
};

main()
{
    Test t, t1, t2, t3;
    return 0;
}
```

Output

```
Constructor executed
Constructor executed
Constructor executed
Constructor executed
Destructor executed
Destructor executed
Destructor executed
Destructor executed
```

C++

```
#include <iostream>
using namespace std;
int count = 0;
class Test {
public:
    Test()
    {
        count++;
        cout << "\n No. of Object created:\t" << count;
    }

    ~Test()
    {
        cout << "\n No. of Object destroyed:\t" << count;
        --count;
    }
};

main()
{
    Test t, t1, t2, t3;
    return 0;
}
```

Output

```
 No. of Object created:    1
 No. of Object created:    2
 No. of Object created:    3
 No. of Object created:    4
 No. of Object destroyed:    4
 No. of Object destroyed:    3
 No. of Object destroyed:    2
 No. of Object destroyed:    1
```

Characteristics of a destructor:-

1. Destructor is invoked automatically by the compiler when its corresponding constructor goes out of scope and releases the memory space that is no longer required by the program.
2. Destructor neither requires any argument nor returns any value therefore it cannot be overloaded.
3. Destructor cannot be declared as static and const;
4. Destructor should be declared in the public section of the program.
5. Destructor is called in the reverse order of its constructor invocation.

Q: What are the functions that are generated by the compiler by default, if we do not provide them explicitly?
Ans: The functions that are generated by the compiler by default if we do not provide them explicitly are:
I. Default constructor
II. Copy constructor
III. Assignment operator
IV. Destructor

Copy Constructor in C++

Difficulty Level : Medium • Last Updated : 16 Mar, 2023

Pre-requisite: Constructor in C++

A **copy constructor** is a member function that initializes an object using another object of the same class. In simple terms, a constructor which creates an object by initializing it with an object of the same class, which has been created previously is known as a **copy constructor**.

Copy constructor is used to initialize the members of a newly created object by copying the members of an already existing object.

Copy constructor takes a reference to an object of the same class as an argument.

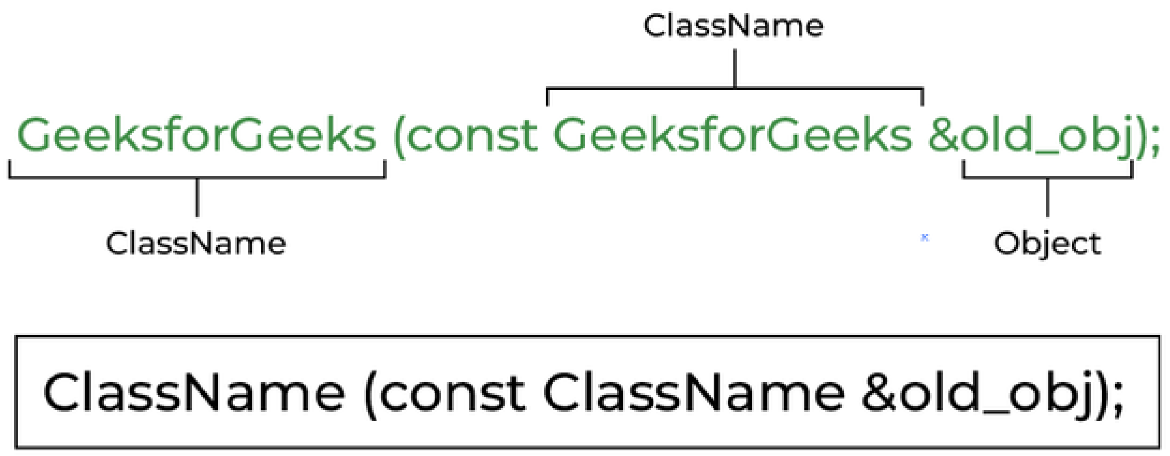
```
Sample(Sample &t)
{
    id=t.id;
}
```

The process of initializing members of an object through a copy constructor is known as copy initialization.

It is also called member-wise initialization because the copy constructor initializes one object with the existing object, both belonging to the same class on a member by member copy basis.

The copy constructor can be defined explicitly by the programmer. If the programmer does not define the copy constructor, the compiler does it for us.

Example:


ClassName
GeeksforGeeks (const GeeksforGeeks &old_obj);
ClassName
Object
ClassName (const ClassName &old_obj);

Syntax of Copy Constructor

C++

```
#include<iostream>
#include<string.h>
using namespace std;
class student
{
    int rno;
    char name[50];
    double fee;
    public:
    student(int,char[],double);
    student(student &t)       //copy constructor
    {
        rno=t.rno;
        strcpy(name,t.name);
        fee=t.fee;
    }
    void display();

};
```

```
    student::student(int no,char n[],double f)
    {
        rno=no;
        strcpy(name,n);
        fee=f;
    }

   void student::display()
    {
        cout<<endl<<rno<<"\t"<<name<<"\t"<<fee;
    }

int main()
{
    student s(1001,"Manjeet",10000);
    s.display();

    student manjeet(s);   //copy constructor called
    manjeet.display();

    return 0;
}
```

C++

```
#include<iostream>
#include<string.h>
using namespace std;
class student
{
    int rno;
    char name[50];
    double fee;
    public:
    student(int,char[],double);
    student(student &t)       //copy constructor (member wise initialization)
    {
        rno=t.rno;
        strcpy(name,t.name);

    }
    void display();
    void disp()
    {
        cout<<endl<<rno<<"\t"<<name;
    }

};
    student::student(int no, char n[],double f)
    {
        rno=no;
        strcpy(name,n);
        fee=f;
    }

   void student::display()
    {
        cout<<endl<<rno<<"\t"<<name<<"\t"<<fee;
    }



int main()
{
    student s(1001,"Manjeet",10000);
    s.display();

    student manjeet(s);   //copy constructor called
    manjeet.disp();

    return 0;
}
```

Characteristics of Copy Constructor

1. The copy constructor is used to initialize the members of a newly created object by copying the members of an already existing object.

2. Copy constructor takes a reference to an object of the same class as an argument.

```
Sample(Sample &t)
{
       id=t.id;
}
```

3. The process of initializing members of an object through a copy constructor is known as ***copy initialization.***

4. It is also called member-wise initialization because the copy constructor initializes one object with the existing object, both belonging to the same class on a member-by-member copy basis.

5. The copy constructor can be defined explicitly by the programmer. If the programmer does not define the copy constructor, the compiler does it for us.

Example:

C++

```cpp
// C++ program to demonstrate the working
// of a COPY CONSTRUCTOR
#include <iostream>
using namespace std;

class Point {
private:
    int x, y;

public:
    Point(int x1, int y1)
    {
        x = x1;
        y = y1;
    }

    // Copy constructor
    Point(const Point& p1)
    {
        x = p1.x;
        y = p1.y;
    }

    int getX() { return x; }
    int getY() { return y; }
};

int main()
{
    Point p1(10, 15); // Normal constructor is called here
    Point p2 = p1; // Copy constructor is called here

    // Let us access values assigned by constructors
    cout << "p1.x = " << p1.getX()
         << ", p1.y = " << p1.getY();
    cout << "\np2.x = " << p2.getX()
         << ", p2.y = " << p2.getY();
    return 0;
}
```

Output

```
p1.x = 10, p1.y = 15
p2.x = 10, p2.y = 15
```

Types of Copy Constructors

1. Default Copy Constructor

An implicitly defined copy constructor will copy the bases and members of an object in the same order that a constructor would initialize the bases and members of the object.

C++

```cpp
// Implicit copy constructor Calling
#include <iostream>
using namespace std;

class Sample {
    int id;

public:
    void init(int x) { id = x; }
    void display() { cout << endl << "ID=" << id; }
};

int main()
{
    Sample obj1;
    obj1.init(10);
    obj1.display();

    // Implicit Copy Constructor Calling
    Sample obj2(obj1); // or obj2=obj1;
    obj2.display();
    return 0;
}
```

Output

```
ID=10
ID=10
```

2. User Defined Copy Constructor

A user-defined copy constructor is generally needed when an object owns pointers or non-shareable references, such as to a file, in which case a destructor and an assignment operator should also be written

C++

```cpp
// Explicitly copy constructor Calling
#include<iostream>
using namespace std;

class Sample
{
    int id;
    public:
    void init(int x)
    {
        id=x;
    }
    Sample(){}  //default constructor with empty body

    Sample(Sample &t)   //copy constructor
    {
        id=t.id;
    }
    void display()
    {
        cout<<endl<<"ID="<<id;
    }
};
int main()
{
    Sample obj1;
    obj1.init(10);
    obj1.display();

    Sample obj2(obj1); //or obj2=obj1;    copy constructor called
    obj2.display();
    return 0;
}
```

Output

```
ID=10
ID=10
```

C++

```
// C++ Programt to demonstrate the student details
#include <iostream>
#include <string.h>
using namespace std;
class student {
    int rno;
    string name;
    double fee;

public:
    student(int, string, double);
    student(student& t) // copy constructor
    {
        rno = t.rno;
        name = t.name;
        fee = t.fee;
    }
    void display();
};
student::student(int no, string n, double f)
{
    rno = no;
    name = n;
    fee = f;
}
void student::display()
{
    cout << endl << rno << "\t" << name << "\t" << fee;
}
int main()
{
    student s(1001, "Ram", 10000);
    s.display();
    student ram(s); // copy constructor called
    ram.display();
    return 0;
}
```

Output

```
1001    Ram    10000
1001    Ram    10000
```

When is the copy constructor called?

In C++, a Copy Constructor may be called in the following cases:

- When an object of the class is returned by value.
- When an object of the class is passed (to a function) by value as an argument.
- When an object is constructed based on another object of the same class.
- 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).

Copy Elision

In copy elision, the compiler prevents the making of extra copies which results in saving space and better the program complexity(both time and space); Hence making the code more optimized.

Example:

C++

```
// C++ program to demonstrate
// the working of copy elision
#include <iostream>
using namespace std;

class GFG {
public:
    void print() { cout << " GFG!"; }
};

int main()
{
    GFG G;
    for (int i = 0; i <= 2; i++) {
        G.print();
        cout<<"\n";
    }
    return 0;
}
```

Output

```
 GFG!
 GFG!
 GFG!
```

Now it is on the compiler to decide what it wants to print, it could either print the above output or it could print case 1 or case 2 below, and this is what ***Return Value Optimization*** is. In simple words, ***RVO*** is a technique that gives the compiler some additional power to terminate the temporary object created which results in changing the observable behavior/characteristics of the final program.

Case 1:

```
GFG!
GFG!
```

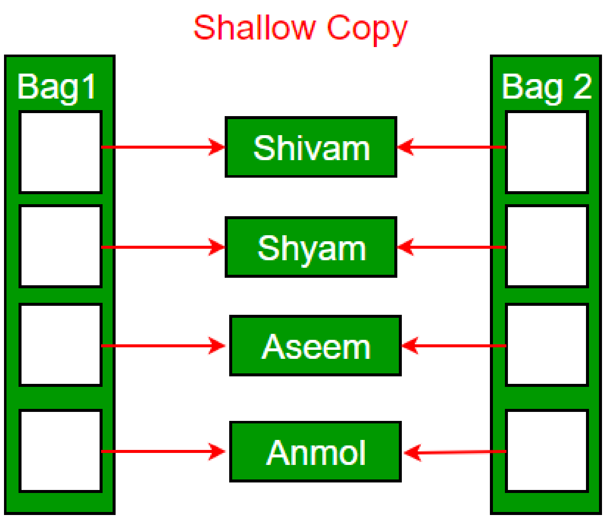
Case 2:

```
GFG!
```

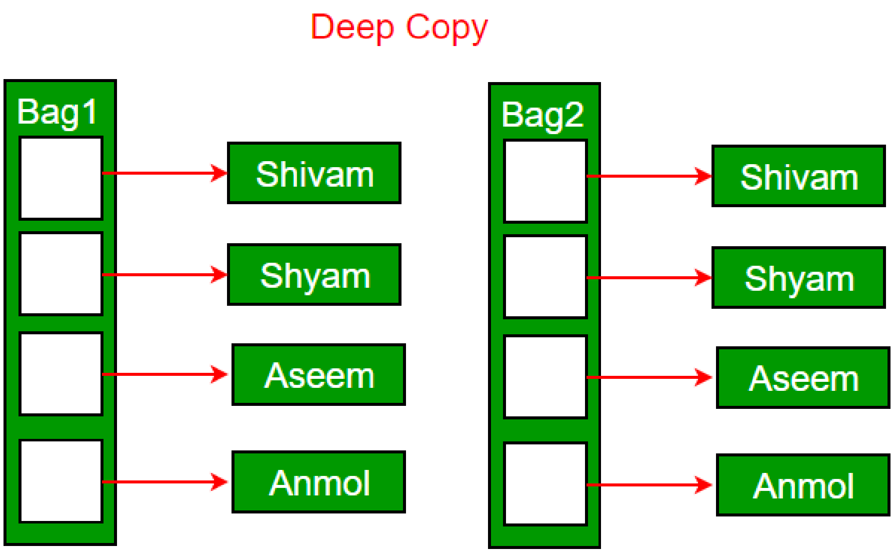
When is a user-defined copy constructor needed?

If we don't define our own copy constructor, the C++ compiler creates a default copy constructor for each class which does a member-wise copy between objects. The compiler-created copy constructor works fine in general. We need to define our own copy constructor only if an object has pointers or any runtime allocation of the resource like *a file* *handle*, a network connection, etc.

The default ***constructor does only shallow copy.***



Deep copy is possible only with a user-defined copy constructor. In a user-defined copy constructor, we make sure that pointers (or references) of copied objects point to new memory locations.



Copy constructor vs Assignment Operator

The main difference between Copy Constructor and Assignment Operator is that the Copy constructor makes a new memory storage every time it is called while the assignment operator does not make new memory storage.

Which of the following two statements calls the copy constructor and which one calls the assignment operator?

```
MyClass t1, t2;
MyClass t3 = t1;  // ----> (1)
t2 = t1;          // -----> (2)
```

A copy constructor is called when a new object is created from an existing object, as a copy of the existing object. The assignment operator is called when an already initialized object is assigned a new value from another existing object. In the above example (1) calls the copy constructor and (2) calls the assignment operator. See this for more details.

Example – Class Where a Copy Constructor is Required

Following is a complete C++ program to demonstrate the use of the Copy constructor. In the following String class, we must write a copy constructor.

Example:

C++

```
// C++ program to demonstrate the
// Working of Copy constructor
#include <cstring>
#include <iostream>
using namespace std;

class String {
private:
    char* s;
    int size;

public:
    String(const char* str = NULL); // constructor
    ~String() { delete[] s; } // destructor
    String(const String&); // copy constructor
    void print()
    {
        cout << s << endl;
    } // Function to print string
    void change(const char*); // Function to change
};

// In this the pointer returns the CHAR ARRAY
// in the same sequence of string object but
// with an additional null pointer '\0'
String::String(const char* str)
{
    size = strlen(str);
    s = new char[size + 1];
    strcpy(s, str);
}

void String::change(const char* str)
```

```
{
    delete[] s;
    size = strlen(str);
    s = new char[size + 1];
    strcpy(s, str);
}

String::String(const String& old_str)
{
    size = old_str.size;
    s = new char[size + 1];
    strcpy(s, old_str.s);
}

int main()
{
    String str1("GeeksQuiz");
    String str2 = str1;

    str1.print(); // what is printed ?
    str2.print();

    str2.change("GeeksforGeeks");

    str1.print(); // what is printed now ?
    str2.print();
    return 0;
}
```

Output

```
GeeksQuiz
GeeksQuiz
GeeksQuiz
GeeksforGeeks
```

What would be the problem if we remove the copy constructor from the above code?

If we remove the copy constructor from the above program, we don’t get the expected output. The changes made to str2 reflect in str1 as well which is never expected.

C++

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

```
class String {
private:
    char* s;
    int size;

public:
    String(const char* str = NULL); // constructor
    ~String() { delete[] s; } // destructor
    void print() { cout << s << endl; }
    void change(const char*); // Function to change
};

String::String(const char* str)
{
    size = strlen(str);
    s = new char[size + 1];
    strcpy(s, str);
}

// In this the pointer returns the CHAR ARRAY
// in the same sequence of string object but
// with an additional null pointer '\0'
void String::change(const char* str) { strcpy(s, str); }

int main()
{
    String str1("GeeksQuiz");
    String str2 = str1;

    str1.print(); // what is printed ?
    str2.print();

    str2.change("GeeksforGeeks");

    str1.print(); // what is printed now ?
    str2.print();
    return 0;
}
```

Output:

```
GeeksQuiz
GeeksQuiz
GeeksforGeeks
GeeksforGeeks
```

Can we make the copy constructor private?

Yes, a copy constructor can be made private. When we make a copy constructor private in a class, objects of that class become non-copyable. This is particularly useful when our class has pointers or dynamically allocated resources. In such situations, we can either write our own copy constructor like the above String example or make a private copy constructor so that users get compiler errors rather than surprises at runtime.

Why argument to a copy constructor must be passed as a reference?

A copy constructor is called when an object is passed by value. Copy constructor itself is a function. So if we pass an argument by value in a copy constructor, a call to the copy constructor would be made to call the copy constructor which becomes a non-terminating chain of calls. Therefore compiler doesn't allow parameters to be passed by value.

Why argument to a copy constructor should be const?

One reason for passing ***const*** reference is, that we should use ***const*** in C++ wherever possible so that objects are not accidentally modified. This is one good reason for passing reference as ***const***, but there is more to it than '*Why argument to a copy constructor should be const?*'

This article is contributed by **Shubham Agrawal**. If you like GeeksforGeeks and would like to contribute, you can also write your article at write.geeksforgeeks.org. See your article appearing on the GeeksforGeeks main page and help other Geeks. Please write comments if you find anything incorrect, or if you want to share more information about the topic discussed above.

420

Related Articles

1. Copy Constructor vs Assignment Operator in C++
2. When is a Copy Constructor Called in C++?
3. When Should We Write Our Own Copy Constructor in C++?
4. Advanced C++ | Virtual Copy Constructor
5. Why copy constructor argument should be const in C++?
6. Different methods to copy in C++ STL | std::copy(), copy_n(), copy_if(), copy_backward()
7. Shallow Copy and Deep Copy in C++

Destructors in C++

Difficulty Level : Easy • Last Updated : 11 Dec, 2022

What is a destructor?

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

- Destructor is also a special member function like constructor. Destructor destroys the class objects created by constructor.
- Destructor has the same name as their class name preceded by a tilde (~) symbol.
- It is not possible to define more than one destructor.
- The destructor is only one way to destroy the object create by constructor. Hence destructor can-not be overloaded.
- Destructor neither requires any argument nor returns any value.
- It is automatically called when object goes out of scope.
- Destructor release memory space occupied by the objects created by constructor.
- In destructor, objects are destroyed in the reverse of an object creation.

The thing is to be noted here, if the object is created by using new or the constructor uses new to allocate memory which resides in the heap memory or the free store, the destructor should use delete to free the memory.

Syntax:

```
Syntax for defining the destructor within the class
~ <class-name>()
{
}



Syntax for defining the destructor outside the class
<class-name>: : ~ <class-name>()
{

}
```

C++

```
// Example:

#include<iostream>
using namespace std;

class Test
{
   public:
         Test()
        {
               cout<<"\n Constructor executed";
        }

        ~Test()
           {
               cout<<"\n Destructor executed";
           }
};
main()
{
     Test t;

    return 0;
}
```

Output

AD

```
Constructor executed
Destructor executed
```

C++

```cpp
// Example:

#include<iostream>
using namespace std;
class Test
{
    public:
        Test()
        {
            cout<<"\n Constructor executed";
        }

        ~Test()
        {
            cout<<"\n Destructor executed";
        }
};

main()
{
    Test t,t1,t2,t3;
    return 0;
}
```

Output

```
Constructor executed
Constructor executed
Constructor executed
Constructor executed
Destructor executed
Destructor executed
Destructor executed
Destructor executed
```

C++

```cpp
// Example:

#include<iostream>
using namespace std;
int count=0;
class Test
```

```
{
    public:
        Test()
        {
            count++;
            cout<<"\n No. of Object created:\t"<<count;
        }

        ~Test()
        {
            cout<<"\n No. of Object destroyed:\t"<<count;
            --count;
        }
};

main()
{
    Test t,t1,t2,t3;
    return 0;
}
```

Output

```
 No. of Object created:    1
 No. of Object created:    2
 No. of Object created:    3
 No. of Object created:    4
 No. of Object destroyed:    4
 No. of Object destroyed:    3
 No. of Object destroyed:    2
 No. of Object destroyed:    1
```

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

```
Note:  destructor can also be called explicitly for an object.
syntax:
object_name.~class_name()
```

How are destructors 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

CPP

```cpp
class String {
private:
    char* s;
    int size;

public:
    String(char*); // constructor
    ~String(); // destructor
};

String::String(char* c)
{
    size = strlen(c);
    s = new char[size + 1];
    strcpy(s, c);
}
String::~String() { delete[] s; }
```

Can there be more than one destructor in a class?

No, there can only one destructor in a class with classname preceded by ~, no parameters and no return type.

When do we need to write a user-defined destructor?

If we do not write our own destructor in class, compiler creates a default destructor for us. The default destructor works fine unless we have dynamically allocated memory or pointer in class. When a class contains a pointer to memory allocated in class, we should write a destructor to release memory before the class instance is destroyed. This must be done to avoid memory leak.

Can a destructor be virtual?

Yes, In fact, it is always a good idea to make destructors virtual in base class when we have a virtual function. See virtual destructor for more details.

You may like to take a quiz on destructors.

Related Articles :

Constructors in C++

Virtual Destructor

Pure virtual destructor in C++

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

263

Related Articles

1. Playing with Destructors in C++
2. C++ Interview questions based on constructors/ Destructors.
3. C++ Error - Does not name a type
4. Execution Policy of STL Algorithms in Modern C++
5. Square Meter (Meter Squared)
6. C++ Program To Print Pyramid Patterns
7. Jagged Arrays in C++
8. Introduction to Parallel Programming with OpenMP in C++
9. Hollow Half Pyramid Pattern Using Numbers
10. 30 OOPs Interview Questions and Answers (2023)

Previous Next

Article Contributed By :

GeeksforGeeks

Default Constructors in C++

Difficulty Level : Easy • Last Updated : 16 Mar, 2023

A constructor without any arguments or with the default value for every argument is said to be the **Default constructor**.

A constructor that has zero parameter list or in other sense, a constructor that accept no arguments is called a zero argument constructor or default constructor.

If default constructor is not defined in the source code by the programmer, then the compiler defined the default constructor implicitly during compilation.

If the default constructor is defined explicitly in the program by the programmer, then the compiler will not defined the constructor implicitly, but it calls the constructor implicitly.

What is the significance of the default constructor?

They are used to create objects, which do not have any specific initial value.

Is a default constructor automatically provided?

If no constructors are explicitly declared in the class, a default constructor is provided automatically by the compiler.

Can a default constructor contain a default argument?

Yes, a constructor can contain default argument with default values for an object.

Will there be any code inserted by the compiler to the user implemented default constructor behind the scenes?

The compiler will implicitly declare the default constructor if not provided by the programmer, will define it when in need. The compiler-defined default constructor is required to do certain initialization of class internals. It will not touch the data members or plain old data types (aggregates like an array, structures, etc…). However, the compiler generates code for the default constructor based on the situation.

Consider a class derived from another class with the default constructor, or a class containing another class object with the default constructor. The compiler needs to insert code to call the default constructors of the base class/embedded object.

C++

```cpp
// CPP program to demonstrate Default constructors
#include <iostream>
using namespace std;

class Base {
public:
    // compiler "declares" constructor
};

class A {
public:
    // User defined constructor
    A() { cout << "A Constructor" << endl; }

    // uninitialized
    int size;
};

class B : public A {
    // compiler defines default constructor of B, and
    // inserts stub to call A constructor

    // compiler won't initialize any data of A
};

class C : public A {
public:
    C()
    {
        // User defined default constructor of C
        // Compiler inserts stub to call A's constructor
        cout << "C Constructor" << endl;

        // compiler won't initialize any data of A
    }
};

class D {
public:
    D()
    {
        // User defined default constructor of D
        // a - constructor to be called, compiler inserts
        // stub to call A constructor
        cout << "D Constructor" << endl;

        // compiler won't initialize any data of 'a'
    }

private:
    A a;
};

// Driver Code
int main()
{
    Base base;

    B b;
    C c;
    D d;

    return 0;
}
```

Output

```
A Constructor
A Constructor
C Constructor
A Constructor
D Constructor
```

C++

```
Example:
#include<iostream>
using namespace std;
class student
{
    int rno;
    char name[50];
    double fee;
    public:
    student()                     //  Explicit Default constructor
    {
        cout<<"Enter the RollNo:";
        cin>>rno;
        cout<<"Enter the Name:";
        cin>>name;
        cout<<"Enter the Fee:";
        cin>>fee;
    }

    void display()
    {
        cout<<endl<<rno<<"\t"<<name<<"\t"<<fee;
    }
};

int main()
{
    student s;
    s.display();
    return 0;
}
```

There are different scenarios in which the compiler needs to insert code to ensure some necessary initialization as per language requirements. We will have them in upcoming posts. Our objective is to be aware of C++ internals, not to use them incorrectly.

C++

```
// CPP code to demonstrate constructor can have default
// arguments
#include <iostream>
using namespace std;
class A {
public:
    int sum = 0;
    A(); // default constructor with no argument
    A(int a, int x = 0) // default constructor with one
                        // default argument
    {
        sum = a + x;
    }
    void print() { cout << "Sum =" << sum << endl; }
};
int main()
{
    // This construct has two arguments. Second argument is
    // initialized with a value of 0 Now we can call the
    // constructor in two possible ways.
    A obj1(10, 20);
    A obj2(5);
    obj1.print();
    obj2.print();
    return 0;
}
```

Output

```
Sum =30
Sum =5
```

Explanation : Here, we have a constructor with two parameter- simple parameter and one default parameter. Now, there are two ways of calling this constructor:

1. First, we can assign values to both the arguments and these values will be passed to the constructor and the default argument x with value 0 will be overridden by value passed while calling (in this case 20). Hence, code will give an output of 30 (as, sum= a+x i.e 10+20= 30).
2. Second way is to not pass any value for the default parameter. If you do so, x will take it's default value 0 as it's final value and calculate a sum of 5 (as, sum = a+x i.e 5+0=5).

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

77

Related Articles

1. When Does Compiler Create Default and Copy Constructors in C++?
2. Inheritance and Constructors in Java
3. When are Constructors Called?
4. Constructors in Java
5. C++ | Constructors | Question 2
6. C++ | Constructors | Question 4
7. C++ | Constructors | Question 5

Private Destructor in C++

Difficulty Level : Medium • Last Updated : 17 Jan, 2023

Destructors with the access modifier as private are known as Private Destructors. Whenever we want to prevent the destruction of an object, we can make the destructor private.

What is the use of private destructor?

Whenever we want to control the destruction of objects of a class, we make the destructor private. For dynamically created objects, it may happen that you pass a pointer to the object to a function and the function deletes the object. If the object is referred after the function call, the reference will become dangling.

Predict the Output of the Following Programs:

CPP

```cpp
// CPP program to illustrate
// Private Destructor
#include <iostream>
using namespace std;

class Test {
private:
    ~Test() {}
};
int main() {}
```

The above program compiles and runs fine. Hence, we can say that: It is **not** a compiler error to create private destructors.

Now, What do you say about the below program?

CPP

```
// CPP program to illustrate
// Private Destructor
#include <iostream>
using namespace std;

class Test {
private:
    ~Test() {}
};
int main() { Test t; }
```

Output

```
prog.cpp: In function ‘int main()’:
prog.cpp:8:5: error: ‘Test::~Test()’ is private
    ~Test() {}
    ^
prog.cpp:10:19: error: within this context
int main() { Test t; }
```

The above program fails in the compilation. The compiler notices that the local variable ‘t’ cannot be destructed because the destructor is private.

Now, What about the Below Program?

CPP

```
// CPP program to illustrate
// Private Destructor
#include <iostream>
using namespace std;

class Test {
private:
    ~Test() {}
};
int main() { Test* t; }
```

The above program works fine. There is no object being constructed, the program just creates a pointer of type "Test *", so nothing is destructed.

Next, What about the below program?

CPP

```cpp
// CPP program to illustrate
// Private Destructor
#include <iostream>
using namespace std;

class Test {
private:
    ~Test() {}
};
int main() { Test* t = new Test; }
```

The above program also works fine. When something is created using dynamic memory allocation, it is the programmer's responsibility to delete it. So compiler doesn't bother.

In the case where the destructor is declared private, an instance of the class can also be created using the malloc() function. The same is implemented in the below program.

CPP

```cpp
// CPP program to illustrate
// Private Destructor

#include <bits/stdc++.h>
using namespace std;

class Test {
public:
    Test() // Constructor
    {
        cout << "Constructor called\n";
    }

private:
    ~Test() // Private Destructor
    {
        cout << "Destructor called\n";
    }
};

int main()
{
    Test* t = (Test*)malloc(sizeof(Test));
    return 0;
}
```

The above program also works fine. However, The below program fails in the compilation. When we call delete, destructor is called.

CPP

```cpp
// CPP program to illustrate
// Private Destructor
#include <iostream>
using namespace std;

class Test {
private:
    ~Test() {}
};

// Driver Code
int main()
{
    Test* t = new Test;
    delete t;
}
```

We noticed in the above programs when a class has a private destructor, only dynamic objects of that class can be created. Following is a way to **create classes with private destructors and have a function as a friend of the class.** The function can only delete the objects.

CPP

```cpp
// CPP program to illustrate
// Private Destructor
#include <iostream>

// A class with private destructor
class Test {
private:
    ~Test() {}

public:
    friend void destructTest(Test*);
};

// Only this function can destruct objects of Test
void destructTest(Test* ptr) { delete ptr; }

int main()
{
    // create an object
    Test* ptr = new Test;

    // destruct the object
    destructTest(ptr);

    return 0;
}
```

Another way to use private destructors is by using **the class instance method**.

C++

```
#include <iostream>

using namespace std;

class parent {
    // private destructor
    ~parent() { cout << "destructor called" << endl; }

public:
    parent() { cout << "constructor called" << endl; }
    void destruct() { delete this; }
};

int main()
{
    parent* p;
    p = new parent;
    // destructor called
    p->destruct();

    return 0;
}
```

Output

```
constructor called
destructor called
```

Must Read: Can a Constructor be Private in C++?

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

156

Related Articles