

CE142: OBJECT ORIENTED PROGRAMMING WITH C++

December – April 2019

Unit – 07

Constructor and Destructors

Constructors

- A constructor is a **special member function** whose task is to initialize the objects of its class.
- It is special because its name is same as the class name.
- The constructor is invoked whenever an object of its associated class is created.
- It is called constructor because it constructs the values of data members of the class.

Constructor - example

```
class add
{
    int m, n ;
    public :
    add (void) ;
    -----
};
add :: add (void)
{
    m = 0; n = 0;
}
```

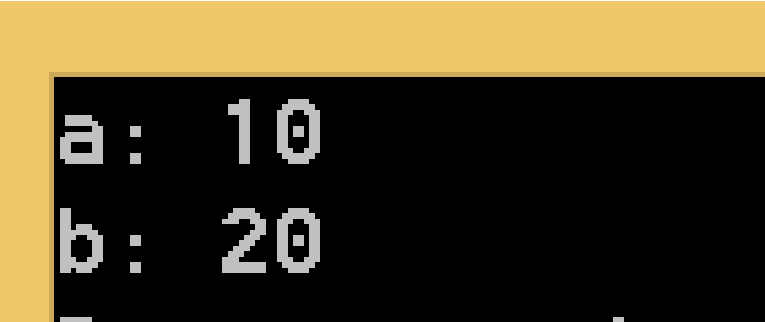
- When a class contains a constructor, it is guaranteed that an object created by the class will be initialized automatically.
- when we writer
 - add a ;
- Not only creates the object a of type add but also initializes its data members m and n to zero.

```
#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;
}
```



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

Constructors

- There is no need to write any statement to invoke the constructor function.
- If a 'normal' member function is defined for zero initialization, we would need to invoke this function for each of the objects separately.
- A constructor that accepts no parameters is called the default constructor.
- The default constructor for class A is `A :: A ()`

Characteristics of Constructors

- declared in the public section.
- invoked automatically when the objects are created.
- No return types, no void and no return values.
- cannot be inherited, though a derived class can call the base class constructor.
- Like other C++ functions, Constructors can have default arguments.
- can not refer to their addresses.

Characteristics of Constructors

- Constructors can not be virtual.
- An object with a constructor (or destructor) can not be used as a member of a union.
- They make ‘implicit calls’ to the operators new and delete when memory allocation is required.
- “When a constructor is declared for a class initialization of the class objects becomes mandatory”

Parameterized Constructors

- It may be necessary to initialize the various data elements of different objects with different values when they are created.
- This is achieved by passing arguments to the constructor function when the objects are created.
- The constructors that can take arguments are called parameterized constructors.

Parameterized Constructors

```
class add
{
    int m, n ;
    public :
    add (int, int) ;
    -----
};
add :: add (int x, int y)
{
    m = x; n = y;
}
```

- When a constructor is parameterized, we must pass the initial values as arguments to the constructor function when an object is declared.
- Two ways Calling:
 - Explicit
 - add sum = add(2,3);
 - Implicit
 - add sum(2,3)
 - Shorthand method

```

#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;
    }
};

```

Output:

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

```

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

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

    return 0;
}

```

Multiple Constructors in a Class

- C + + permits to use more than one constructors in a single class.
- Add() ; // No arguments
- Add (int, int) ; // Two arguments

Multiple Constructors in a Class

```
class add
{
    int m, n ;

    public :

        add ( )
        {
            m = 0 ;
            n = 0 ;
        }

//constructor receives no
arguments.
```

```
add (int a, int b)
{
    m = a ;
    n = b ;
}

add (add & i)
{
    m = i.m ;
    n = i.n ;
}

};
```

This constructor receives two integer arguments.

This constructor receives one add object as an argument.

Multiple Constructors in a Class

```
class add
{
    int m, n ;
public :
    add ( )
        {m = 0 ; n = 0 ;}
    add (int a, int b)
        {m = a ; n = b ;}
    add (add & i)
        {m = i.m ; n = i.n ;}
};
```

- Add a1;
Would automatically invoke the first constructor and set both m and n of a1 to zero.
- Add a2(10,20);
Would call the second constructor which will initialize the data members m and n of a2 to 10 and 20 respectively.

Multiple Constructors in a Class

```
class add
{
    int m, n ;
public :
    add ( ) {m = 0 ; n = 0 ;}
    add (int a, int b)
        {m = a ; n = b ;}
    add (add & i)
        {m = i.m ; n = i.n ;}
};
```

- Add a3(a2);

Would invoke the third constructor which copies the values of a2 into a3.

This type of constructor is called the “copy constructor”.

- Construction Overloading

More than one constructor function is defined in a class.

Multiple Constructors in a Class

```
class complex
{
    float x, y ;
public :
    complex ( ) { }
    complex (float a)
        { x = y = a ; }
    complex (float r, float i)
        { x = r ; y = i }
    -----
};
```

- `complex () { }`

This contains the empty body and does not do anything.

This is used to create objects without any initial values.

Multiple Constructors in a Class

- C + + compiler has an implicit constructor which creates objects, even though it was not defined in the class.
- This works well as long as we do not use any other constructor in the class.
- However, once we define a constructor, we must also define the “do-nothing” implicit constructor.


```

#include <iostream>
using namespace std;

class Area
{
private:
    int length;
    int breadth;

public:
    // Constructor with no arguments
    Area(): length(5), breadth(2) { }
    // Constructor with two arguments
    Area(int l, int b): length(l), breadth(b) { }
    void GetLength()
    {
        cout << "Enter length and breadth respectively: ";
        cin >> length >> breadth;
    }
    int AreaCalculation() { return length * breadth; }
    void DisplayArea(int temp)
    {
        cout << "Area: " << temp << endl;
    }
};

```

```

int main()
{
    Area A1, A2 (2, 1);
    int temp;

    cout << "Default Area when no argument is passed." << endl;
    temp = A1.AreaCalculation();
    A1.DisplayArea(temp);

    cout << "Area when (2,1) is passed as argument." << endl;
    temp = A2.AreaCalculation();
    A2.DisplayArea(temp);

    return 0;
}

```

Output:

Default Area when no argument is passed.

Area: 10

Area when (2,1) is passed as argument.

Area: 2

Constructors with Default Arguments

- It is possible to define constructors with default arguments.
- Consider `complex (float real, float imag = 0);`
 - The default value of the argument `imag` is zero.
 - `complex C1 (5.0)` assigns the value 5.0 to the real variable and 0.0 to `imag`.
 - `complex C2(2.0,3.0)` assigns the value 2.0 to `real` and 3.0 to `imag`.

Constructors with Default Arguments

- `A :: A ()` \Rightarrow Default constructor
- `A :: A (int = 0)` \Rightarrow Default argument constructor
- The default argument constructor can be called with either one argument or no arguments.
- When called with no arguments, it becomes a default constructor.

Dynamic Initialization of Objects

- Providing initial value to objects at run time.
- Advantage – We can provide various initialization formats, using overloaded constructors.
- This provides the flexibility of using different format of data at run time depending upon the situation.



```

#include <iostream>
using namespace std;
class simple_interest
{
    float principle , time, rate ,interest;

public:
    simple_interest (float a, float b, float c) {
        principle = a;
        time =b;
        rate = c;
    }
    void display ( ) {
        interest =(principle* rate* time)/100;
        cout<<"interest ="<<interest ;
    }
};

```

```
int main() {  
  
    float p,r,t;  
  
    cout<<"principle amount, time and rate"<<endl;  
    cout<<"2000          7.5          2"<<endl;  
  
    simple_interest s1(2000,7.5,2); //dynamic initialization  
  
    s1.display();  
  
    return 1;  
}
```

Output:

```
principle amount, time and rate  
2000          7.5          2  
    interest =300
```

Copy Constructor

- A copy constructor is used to declare and initialize an object from another object.
- `integer (integer & i) ;`
- `integer I 2 (I 1) ;` or `integer I 2 = I 1 ;`
- The process of initializing through a copy constructor is known as copy initialization.

Copy Constructor

- The statement

I 2 = I 1;

will not invoke the copy constructor.

- If I 1 and I 2 are objects, this statement is legal and assigns the values of I 1 to I 2, member-by-member.
- A reference variable has been used as an argument to the copy constructor.
- We cannot pass the argument by value to a copy constructor.

```

#include<iostream>
using namespace std;
class example
{
    private:
        int x;
    public:
        //parameterized constructor to initialize l and b
        example (int a)
        {
            x = a;
        }
        //copy constructor with reference object argument
        example( example &b)
        {
            x = b.x;
        }
        //function to display
        int display( )
        {
            return x;
        }
};

```

```

int main()
{
    //initializing the data members of object 'c' implicitly
    example c1(2);
    //copy constructor called
    example c2(c1);
    example c3 = c1;
    example c4 = c2;
    cout << "example c1 = " << c1.display() << endl;
    cout << "example c2 = " << c2.display() << endl;
    cout << "example c3 = " << c3.display() << endl;
    cout << "example c4 = " << c4.display() << endl;
    return 0;
}

```

Output:

```

example c1 = 2
example c2 = 2
example c3 = 2
example c4 = 2

```

Dynamic Constructors

- The constructors can also be used to allocate memory while creating objects.
- This will enable the system to allocate the right amount of memory for each object when the objects are not of the same size.
- Allocation of memory to objects at the time of their construction is known as dynamic construction of objects.
- The memory is created with the help of the new operator.

```

#include <iostream>
using namespace std;
class dyncons
{
    int * p;
public:
    dyncons ()
    {
        p=new int;
        *p=10;
    }

    dyncons (int v)
    {
        p=new int;
        *p=v;
    }
    int dis ()
    {
        return (*p) ;
    }
};

```

```

int main ()
{
    dyncons o, o1(9);
    cout<<"The value of object o's p is:";
    cout<<o.dis ();
    cout<<"\nThe value of object 01's p is:";
    cout<<o1.dis ();
    return 0;
}

```

Output:

```

The value of object o's p is:10
The value of object 01's p is:9

```

Destructors

- A destructor is used to destroy the objects that have been created by a constructor.
- Like constructor, the destructor is a member function whose name is the same as the class name but is preceded by a tilde.
- e.g. `~integer () { }`
- A destructor never takes any argument nor does it return any value.

Destructors

- It will be invoked implicitly by the compiler upon exit from the program – or block or function as the case may be – to clean up storage that is no longer accessible.
- It is a good practice to declare destructors in a program since it releases memory space for further use.
- Whenever new is used to allocate memory in the constructor, we should use delete to free that memory.

```

#include<iostream>
using namespace std;

class Marks
{
public:
    int maths;
    int science;

    //constructor
    Marks() {
        cout << "Inside Constructor"<<endl;
        cout << "C++ Object created"<<endl;
    }

    //Destructor
    ~Marks() {
        cout << "Inside Destructor"<<endl;
        cout << "C++ Object destroyed"<<endl;
    }
};

```

Output:

```

Inside Constructor
C++ Object created

```

```

Inside Constructor
C++ Object created

```

```

Inside Destructor
C++ Object destroyed

```

```

Inside Destructor
C++ Object destroyed

```


- The primary use of destructors is to free up the memory reserved by an object before it get destroyed. Here it is shown that how destructor releases the memory allocated to an object.

```

#include<iostream>
using namespace std;
class Marks
{
    int *a;
public:

    //constructor
    Marks(int size) {
        a=new int[size];
        cout<<"\n\n    Constructor msg: Integer array of size "<< size<<" created...";
    }

    //Destructor
    ~Marks() {
        delete a;
        cout << "\n\n    Destructed msg: Freed up the memory allocated for integer array"<<endl;
        cout<<endl;
    }
};

int main( )
{
    Marks m1(10);
    return 0;
}

```

Output:

```
Constructor msg: Integer array of size 10 created...
```

```
Destructed msg: Freed up the memory allocated for integer array
```

Presentation Prepared By:



Ms. Khushi Patel

Contact us:

dweepnagarg.ce@charusat.ac.in
parthgoel.ce@charusat.ac.in
hardikjayswal.it@charusat.ac.in
dipakramoliya.ce@charusat.ac.in
krishnapatel.ce@charusat.ac.in
khushipatel.ce@charusat.ac.in

Subject Teachers:



Ms. Dweepna Garg
Subject Coordinator



Mr. Parth Goel
<https://parthgoelblog.wordpress.com>



Mr. Hardik Jayswal



Ms. Krishna Patel

Thank You