Constructor Destructor

Sunday, June 24, 2018 20:37

Constructor and Destructor:

1. **Copy and Move**

Copy	**Move**
1. After a copy, 2 objects must have the same value	1. After a move, Source is not required to have its original value

2. **Constructors in C++:**

   a. They are used to initialize class objects.
   b. Class Members using member initialization lists are initialized in the same order as they are declared
   c. They don't have a return type.
   d. Automatically called when an object is created

   e. Overloading Constructor: based on
      i. Number of input parameters
      ii. Type of input parameters

   f. Default Constructor:
      1) If no C++ constructor is specified, a default constructor is created implicitly
      2) Has empty body
      3) Has no parameters
      4) Only declares the object, but does not initialize it

   g. Can be used for parameters initialization or Member initialization using colons

3. **Summary of various constructors:**

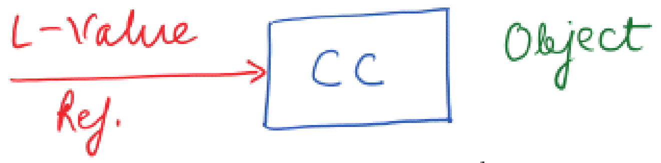
Member function	typical form for class C:
Default constructor	C::C();
Destructor	C::~C();
Copy constructor	C::C (const C&);
Copy assignment	C& operator= (const C&);
Move constructor	C::C (C&&);
Move assignment	C& operator= (C&&);

A. Implicitly defined:

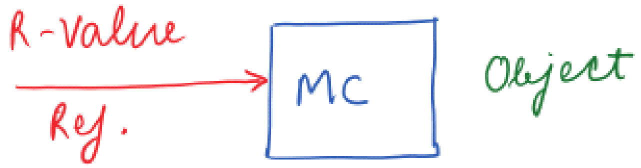
Default Constructor	1. If no user-defined Default Constructor exists, compiler will always generate one provided, there is no user-defined Copy Constructor
Copy Constructor	1. If no user-defined Copy Constructor exists, compiler will always generate one
Destructor	1. If no user-defined Destructor exists, Compiler will always create one
i. **Move Constructor**	1. If no user defined Move Constructor exists, AND there are: 1. NO user-declared Copy Constructors 2. NO user-declared Copy Assignment operators 3. NO user-defined Move Assignment operators 4. No user-defined Destructors 2. Then, compiler will generate an implicit one
Copy Assignment	Complex Rules
Move Assignment	Complex Rules

C. During dynamic memory allocation in class, it is very important to write

User-defined Copy Constructor	1. To prevent shallow copy of pointers during CC operation
User-Defined Destructor	1. To prevent memory leakages when object goes out of scope

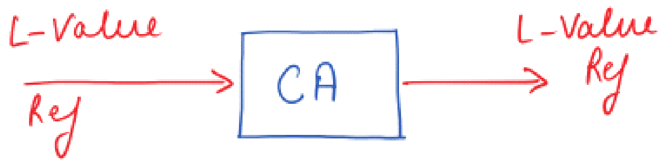


```
C:: C(const C &)
```

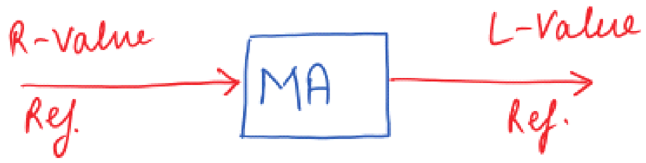


```
C::C( C &&)
```

⟹ Constructors don't have a return type.



```
C& operator = (const C &)
```



```
C & operator = ( C &&)
```

⟹ Assignment operators return a L-Value reference to the object.

1. Move Constructor
2. Move Assignment

3. Copy Constructor in C++:

a. Initializes an object using another object of same class

b. Instances where a copy constructor is called:

   i.

Object is returned by value
Object is passed as call by value to function
Object constructed using another object of same class
Compiler generates a temporary object

c. Examples:

   My Class foo;
   MyClass bar (foo); *// object initialization: copy constructor called*
   MyClass baz = foo; *// object initialization: copy constructor called*
   Foo = bar; // Object already declared. Here is initialized. This is not Copy constructor. But, this is copy assignment.

d. Implicit Copy Constructor:

   i. Automatically defined
   ii. Performs a shallow copy of the provided object
   iii. Shallow copy simply copies all members of object to another object
   iv. Problems with shallow copy:
      1) If an object carries a pointer as a class member, shallow copy would copy the pointer value to another object
      2) On deleting any such object, pointer will be deleted and memory freed using the destructor.
      3) When the second object which carries the same object pointer tries to access this memory, segfault occurs

e. User Defined Copy Constructor:

   i. A User defined copy constructor is a requirement if :
      1) The object carries a pointer
      2) The object allocates resource on run-time

   ii. User defined copy constructor generally should perform a deep copy:
      1) Deep Copy copies the content of the pointer and not the pointer itself

f. Arguments to copy constructor should be :
   i. Passed by reference:
      1) because if they are passed by value, a call to copy constructor would copy constructor again. This would form an infinite chain of copy constructor calls.
   ii. *Passed as const objects*:
      1) as they cannot be modified, and even should not be modified, as in a copy constructor, we just want to copy the values of the object.

4. Destructors in C++:

a. Called Automatically when:

   i. End of Scope (executed after the return statement)
      1) Function ends
      2) Program ends
      3) Block containing local variables ends

   ii. Delete operator is called

b. Destructor has same name as a class
c. Destructor doesn't return anything
d. Destructor doesn't take any argument

e. Default Destructor:
   i. If no destructor is implicitly defined

f. Explicit destructor should always be written if dynamic memory is allocated to free the memory

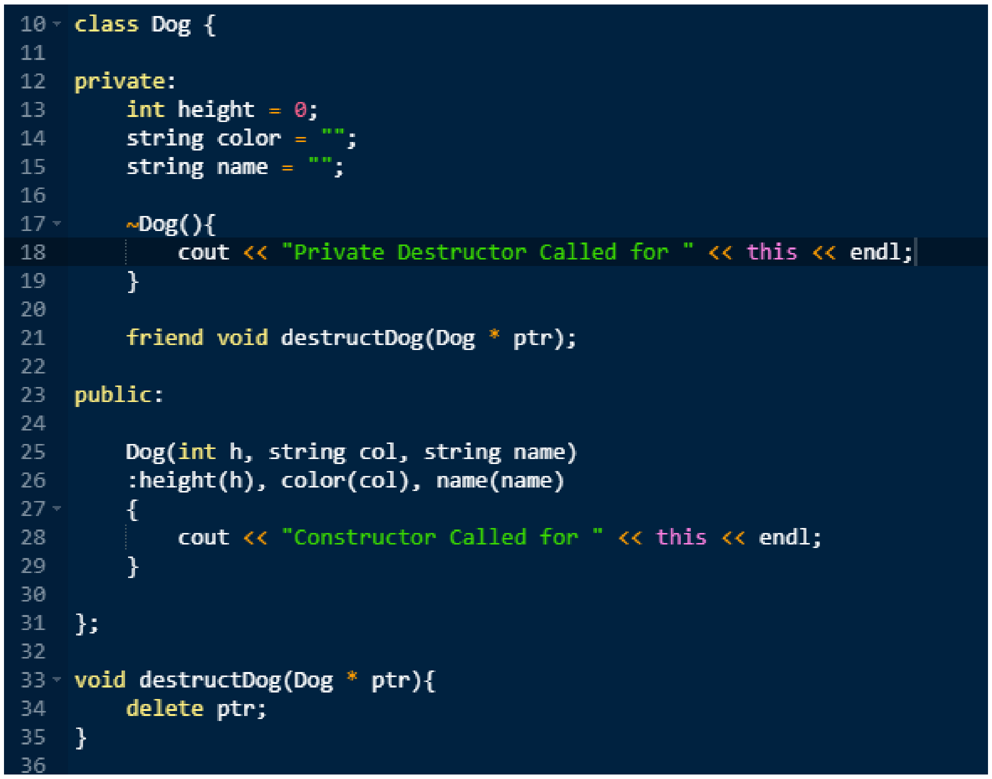
g. Base and Member Destructors:
   i. Constructor constructs an object from bottom up:
      1) First, invokes the Base class constructor
      2) Then, invokes member constructors
      3) Finally, executes its own body

   ii. Destructor tears down the object in reverse order:
      1) First, Destructor executes its own body
      2) Then, it invokes member destructors
      3) Finally, invokes Base Class Destructors

5. Private Destructor:

a. Normally in a destructor, we write delete for any dynamically created objects.
b. However, destructors are automatically called when a function or program ends, and hence destructor will be called - deleting our pointer to object.
c. To avoid such a scenario, we use private destructors.
d. Once we have created a Private Destructor, we cannot create an object of that Class directly. We can only assign a pointer to an object of that class

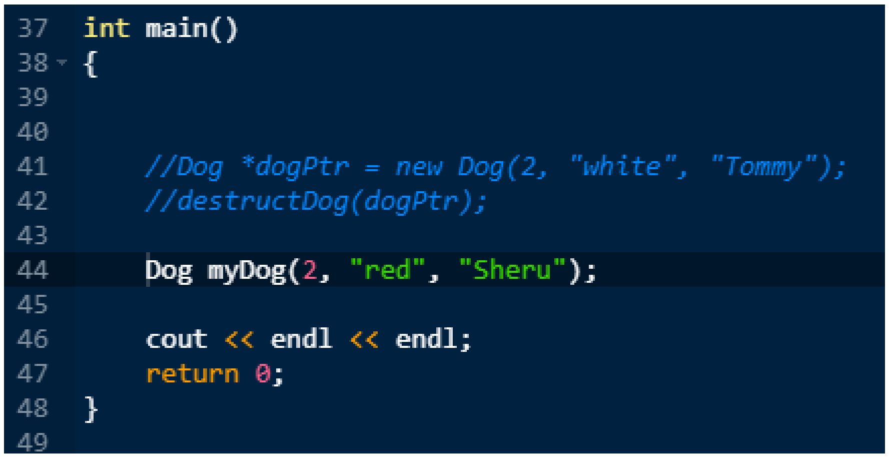
Class Declaration	



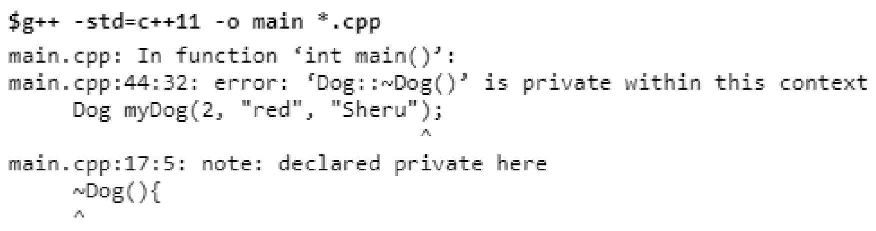
```
10  class Dog {
11
12  private:
13      int height = 0;
14      string color = "";
15      string name = "";
16
17      ~Dog(){
18          cout << "Private Destructor Called for " << this << endl;
19      }
20
21      friend void destructDog(Dog * ptr);
22
23  public:
24
25      Dog(int h, string col, string name)
26      :height(h), color(col), name(name)
27      {
28          cout << "Constructor Called for " << this << endl;
29      }
30
31  };
32
33  void destructDog(Dog * ptr){
34      delete ptr;
35  }
36
```

Direct object for Class

1. Unable to create a direct object for the Class
2. Compilation Error

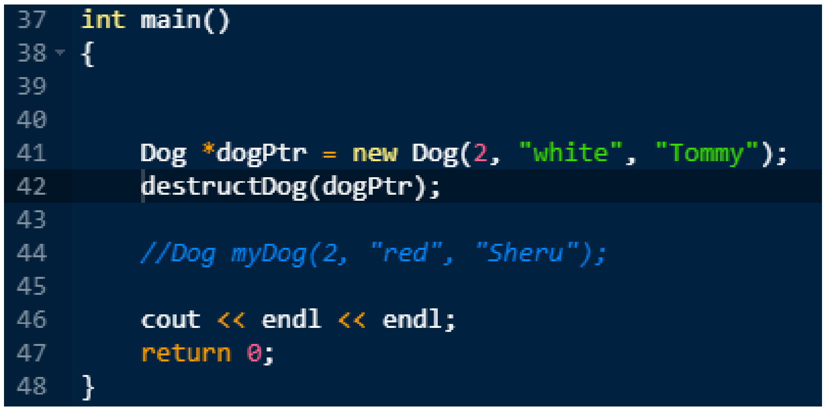
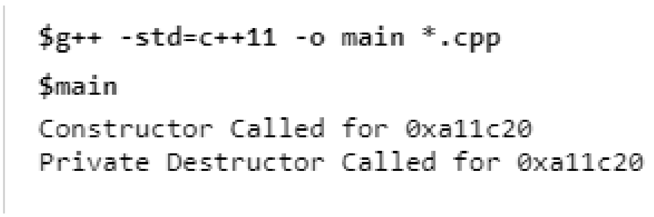


```
37  int main()
38  {
39
40
41      //Dog *dogPtr = new Dog(2, "white", "Tommy");
42      //destructDog(dogPtr);
43
44      Dog myDog(2, "red", "Sheru");
45
46      cout << endl << endl;
47      return 0;
48  }
49
```



```
$g++ -std=c++11 -o main *.cpp
main.cpp: In function ‘int main()’:
main.cpp:44:32: error: ‘Dog::~Dog()’ is private within this context
     Dog myDog(2, "red", "Sheru");
                                ^
main.cpp:17:5: note: declared private here
     ~Dog(){
     ^
```

Pointer for Class

Object	 

```
int main()
{


    Dog *dogPtr = new Dog(2, "white", "Tommy");
    destructDog(dogPtr);

    //Dog myDog(2, "red", "Sheru");

    cout << endl << endl;
    return 0;
}
```

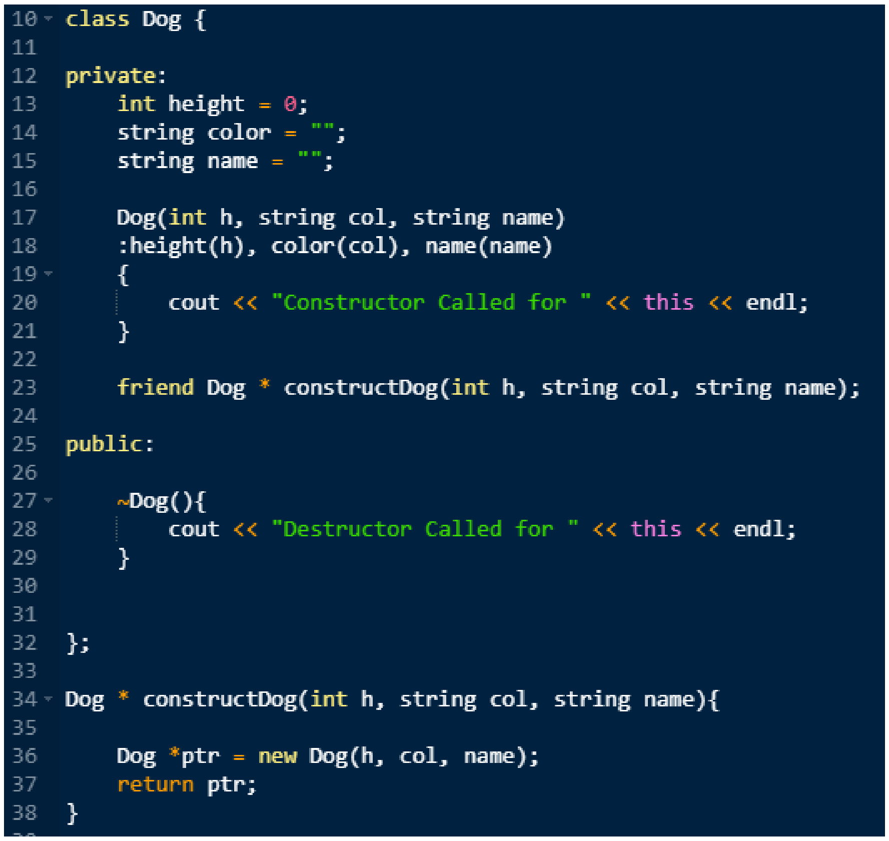
```
$g++ -std=c++11 -o main *.cpp
$main
Constructor Called for 0xa11c20
Private Destructor Called for 0xa11c20
```

e. In this case, only private destructors can delete the pointer to objects, and free the heap memory.
f. After creating a private destructor, we remove the delete command from the actual destructor and make it private.

6. Private Constructor:

a. We can make a Constructor as private for Class
b. Here, we are restricting how we can create an object
c. In fact, we use the same concept in Singleton
d. Example:

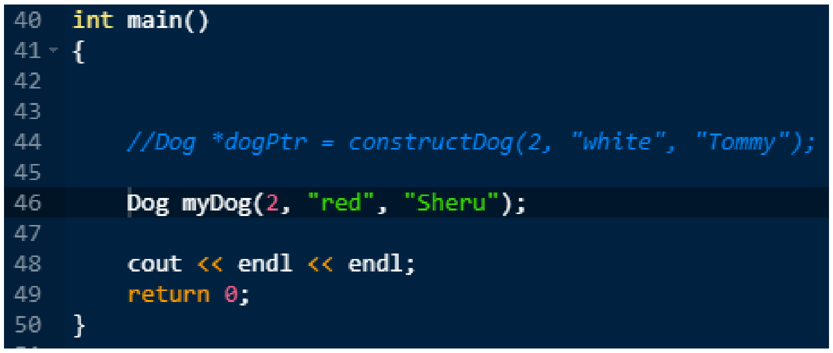
Class	



```cpp
class Dog {

private:
    int height = 0;
    string color = "";
    string name = "";

    Dog(int h, string col, string name)
    :height(h), color(col), name(name)
    {
        cout << "Constructor Called for " << this << endl;
    }

    friend Dog * constructDog(int h, string col, string name);

public:

    ~Dog(){
        cout << "Destructor Called for " << this << endl;
    }


};

Dog * constructDog(int h, string col, string name){

    Dog *ptr = new Dog(h, col, name);
    return ptr;
}
```

Object Instantiation

1. Trying to create an object directly through instantiation

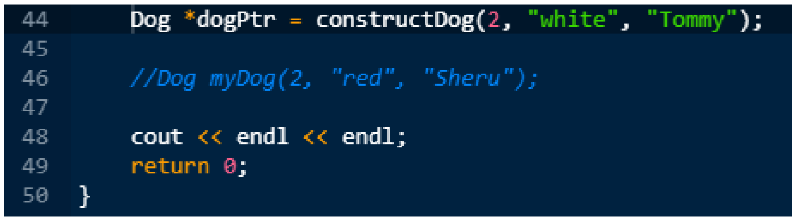
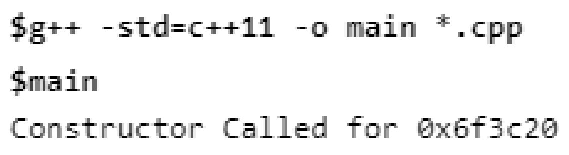


```cpp
int main()
{


    //Dog *dogPtr = constructDog(2, "white", "Tommy");

    Dog myDog(2, "red", "Sheru");

    cout << endl << endl;
    return 0;
}
```

2. As expected, this gives a Compilation Error as the Constructor is private

```
$g++ -std=c++11 -o main *.cpp
main.cpp: In function ‘int main()’:
main.cpp:46:32: error: ‘Dog::Dog(int, std::__cxx11::string, std::
     Dog myDog(2, "red", "Sheru");
                                ^
main.cpp:17:5: note: declared private here
     Dog(int h, string col, string name)
     ^~~
```

Object

Instantiation through Pointer	1.
	
	`    Dog *dogPtr = constructDog(2, "white", "Tommy");`
	
	`    //Dog myDog(2, "red", "Sheru");`
	
	`    cout << endl << endl;`
	`    return 0;`
	`}`
	2. This runs successfully as we are creating object through a friend function
	
	`$g++ -std=c++11 -o main *.cpp`
	`$main`
	`Constructor Called for 0x6f3c20`

After making the Constructor or Destructor - private, we cannot create the object through its normal Constructor. Afterwards, it can only be constructed using pointers.

7. Virtual Destructor:
   a. Done in Virtual

8. Playing with Destructors in C++:

   a. Destructors are called automatically on program, block, or function exits
   b. If such a block also contains return statements, destructor is called after the return statement

   c. Code Examples

d.

`class A`
`{`
`public:`
`    ~A()`
`    {`
`        i=10;`
`    }`
`};`
 	
`int foo()`
`{`
`    i = 3;`
`    A ob;`
`    return i;`
`}`
	`class A`
	`{`
	`public:`
	`    ~A()`
	`    {`
	`        i = 10;`
	`    }`
	`};`
	
	`int& foo()`
	`{`
	`    i = 3;`
	`    A ob;`
	`    return i;`
	`}`
		`class A`
		`{`
		`public:`
		`    ~A()`
		`    {`
		`        i = 10;`
		`    }`
		`};`
		
		`int foo()`
		`{`
		`    i = 3;`
		`    {`
		`        A ob;`
		`    }`
		`    return i;`
		`}`

int main()
{
    cout << "i = "
<< foo() << endl;
    return 0;
}
	int main()
	{
	    cout << "i = "
	<< foo() << endl;
	    return 0;
	}
		int main()
		{
		    cout << "i = " << foo()
		<< endl;
		    return 0;
		}

Returns I = 3 Destructor called after the return statement	Returns I = 10 Destructor called after return statement here as well. But, return statement is by reference. So 10 is returned	Returns I = 10 Destructor called after the block ends . So I is updated to 10, and then it is returned.

9. When does compiler create a default constructor and copy constructor in C++?:

   a. If we create any explicit constructor - whether it be a copy constructor, compiler will not create an implicit default constructor.
   b. In such a case,
      i. if we write a user defined copy constructor, it is imperative for us to write a normal constructor first.
      ii. If we don't write a normal constructor, and write only a user defined copy constructor, it would be a compilation error while creating object of that class.
   c. Reverse is not true. If we write a user defined default constructor, compiler will still generate an implicit copy constructor for us.

10. Is it possible to call constructor and destructor explicitly:

    a. Yes, programmer can explicitly call constructor and destructor explicitly