CL-217 OBJECT ORIENTED PROGRAMMING LAB

LAB MANUAL 8

INSTRUCTOR: MUHAMMAD HAMZA

SEMESTER SPRING 2020

C++ Copy Constructor

The copy constructor is a constructor which creates an object by initializing it with an object of the same class, which has been created previously.

If a copy constructor is not defined in a class, the compiler itself defines one.If the class has pointer variables and has some dynamic memory allocations, then it is a must to have a copy constructor. The most common form of copy constructor is shown here −

```
classname (const classname &obj) {
   // body of constructor
}
```

Here, obj is a reference to an object that is being used to initialize another object.

Live Demo

```
#include <iostream>

using namespace std;

class Line {

   public:
      int getLength( void );
      Line( int len );             // simple constructor
      Line( const Line &obj);  // copy constructor
      ~Line();                     // destructor

   private:
      int *ptr;
};

// Member functions definitions including constructor
Line::Line(int len) {
   cout << "Normal constructor allocating ptr" << endl;
```

```
   // allocate memory for the pointer;
   ptr = new int;
   *ptr = len;
}

Line::Line(const Line &obj) {
   cout << "Copy constructor allocating ptr." << endl;
   ptr = new int;
   *ptr = *obj.ptr; // copy the value
}

Line::~Line(void) {
   cout << "Freeing memory!" << endl;
   delete ptr;
}

int Line::getLength( void ) {
   return *ptr;
}

void display(Line obj) {
   cout << "Length of line : " << obj.getLength() <<endl;
}

// Main function for the program
int main() {
   Line line(10);

   display(line);

   return 0;
}
```

When the above code is compiled and executed, it produces the following result −

```
Normal constructor allocating ptr
Copy constructor allocating ptr.
Length of line : 10
Freeing memory!
Freeing memory!
```

Let us see the same example but with a small change to create another object using existing object of the same type −

Live Demo

```
#include <iostream>

using namespace std;

class Line {
   public:
      int getLength( void );
      Line( int len );             // simple constructor
      Line( const Line &obj);  // copy constructor
      ~Line();                     // destructor

   private:
      int *ptr;
};

// Member functions definitions including constructor
Line::Line(int len) {
   cout << "Normal constructor allocating ptr" << endl;

   // allocate memory for the pointer;
   ptr = new int;
   *ptr = len;
}

Line::Line(const Line &obj) {
   cout << "Copy constructor allocating ptr." << endl;
   ptr = new int;
   *ptr = *obj.ptr; // copy the value
}

Line::~Line(void) {
   cout << "Freeing memory!" << endl;
   delete ptr;
}
```

```
int Line::getLength( void ) {
   return *ptr;
}

void display(Line obj) {
   cout << "Length of line : " << obj.getLength() <<endl;
}

// Main function for the program
int main() {

   Line line1(10);

   Line line2 = line1; // This also calls copy constructor

   display(line1);
   display(line2);

   return 0;
}
```

When the above code is compiled and executed, it produces the following result −

```
Normal constructor allocating ptr
Copy constructor allocating ptr.
Copy constructor allocating ptr.
Length of line : 10
Freeing memory!
Copy constructor allocating ptr.
Length of line : 10
Freeing memory!
Freeing memory!
Freeing memory!
```

When is a copy constructor called?

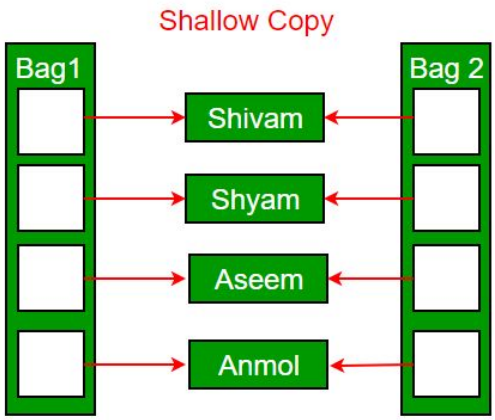
1. when an existing object is assigned an object of it own class
   MyClass A,B; A = new MyClass(); B=A; //copy constructor called

2. if a functions receives as argument, passed by value, an object of a class void foo(MyClass a); foo(a); //copy constructor invoked
3. when a function returns (by value) an object of the class MyClass foo () { MyClass temp; .... return temp; //copy constructor called }

When is a user-defined copy constructor needed?

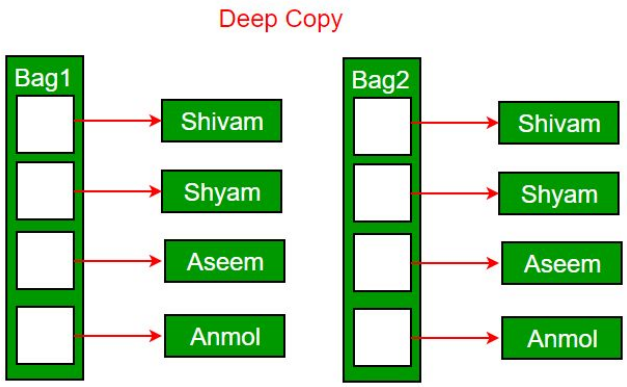
If we don't define our own copy constructor, the C++ compiler creates a default copy constructor for each class which does a memberwise copy between objects. The compiler created copy constructor works fine in general. We need to define our own copy constructor only if an object has pointers or any runtime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

Default constructor does only shallow copy.



Deep copy is possible only with a user defined copy constructor.

In user defined copy constructor, we make sure that pointers (or references) of copied object point to new memory locations.


Deep Copy
Bag1
Shivam
Shyam
Aseem
Anmol
Bag2
Shivam
Shyam
Aseem
Anmol

Friend Function and Friend Class

Friend functions

In principle, private and protected members of a class cannot be accessed from outside the same class in which they are declared. However, this rule does not apply to *"friends"*.

Friends are functions or classes declared with the friend keyword.

A non-member function can access the private and protected members of a class if it is declared a *friend* of that class. That is done by including a declaration of this external function within the class, and preceding it with the keyword friend:

```
// friend functions
#include <iostream>
using namespace std;

class Rectangle {
    int width, height;
  public:
    Rectangle() {}
    Rectangle (int x, int y) : width(x),
height(y) {}
    int area() {return width * height;}
    friend Rectangle duplicate (const
Rectangle&);
};

Rectangle duplicate (const Rectangle&
param)
{
  Rectangle res;
  res.width = param.width*2;
  res.height = param.height*2;
  return res;
}

int main () {
```

```
24
```

```
  Rectangle foo;
  Rectangle bar (2,3);
  foo = duplicate (bar);
  cout << foo.area() << '\n';
  return 0;
}
```

The duplicate function is a *friend* of class Rectangle. Therefore, function duplicate is able to access the members width and height (which are private) of different objects of type Rectangle. Notice though that neither in the declaration of duplicate nor in its later use in main, function duplicate is considered a member of class Rectangle. It isn't! It simply has access to its private and protected members without being a member.

Typical use cases of friend functions are operations that are conducted between two different classes accessing private or protected members of both.

Friend classes

Similar to friend functions, a friend class is a class whose members have access to the private or protected members of another class:

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

class Square;

class Rectangle {
  int width, height;
 public:
  int area ()
    {return (width * height);}
  void convert (Square a);
};

class Square {
 friend class Rectangle;
 private:
  int side;
 public:
  Square (int a) : side(a) {}
};

void Rectangle::convert (Square a) {
 width = a.side;
 height = a.side;
}

int main () {
 Rectangle rect;
 Square sqr (4);
 rect.convert(sqr);
 cout << rect.area();
 return 0;
```

```
16
```

```
}
```

In this example, class Rectangle is a friend of class Square allowing Rectangle's member functions to access private and protected members of Square. More concretely, Rectangle accesses the member variable Square::side, which describes the side of the square.

There is something else new in this example: at the beginning of the program, there is an empty declaration of class Square. This is necessary because class Rectangle uses Square (as a parameter in member convert), and Square uses Rectangle (declaring it a friend).

Friendships are never corresponded unless specified: In our example, Rectangle is considered a friend class by Square, but Square is not considered a friend by Rectangle. Therefore, the member functions of Rectangle can access the protected and private members of Square but not the other way around. Of course, Square could also be declared friend of Rectangle, if needed, granting such an access.

Another property of friendships is that they are not transitive: The friend of a friend is not considered a friend unless explicitly specified.