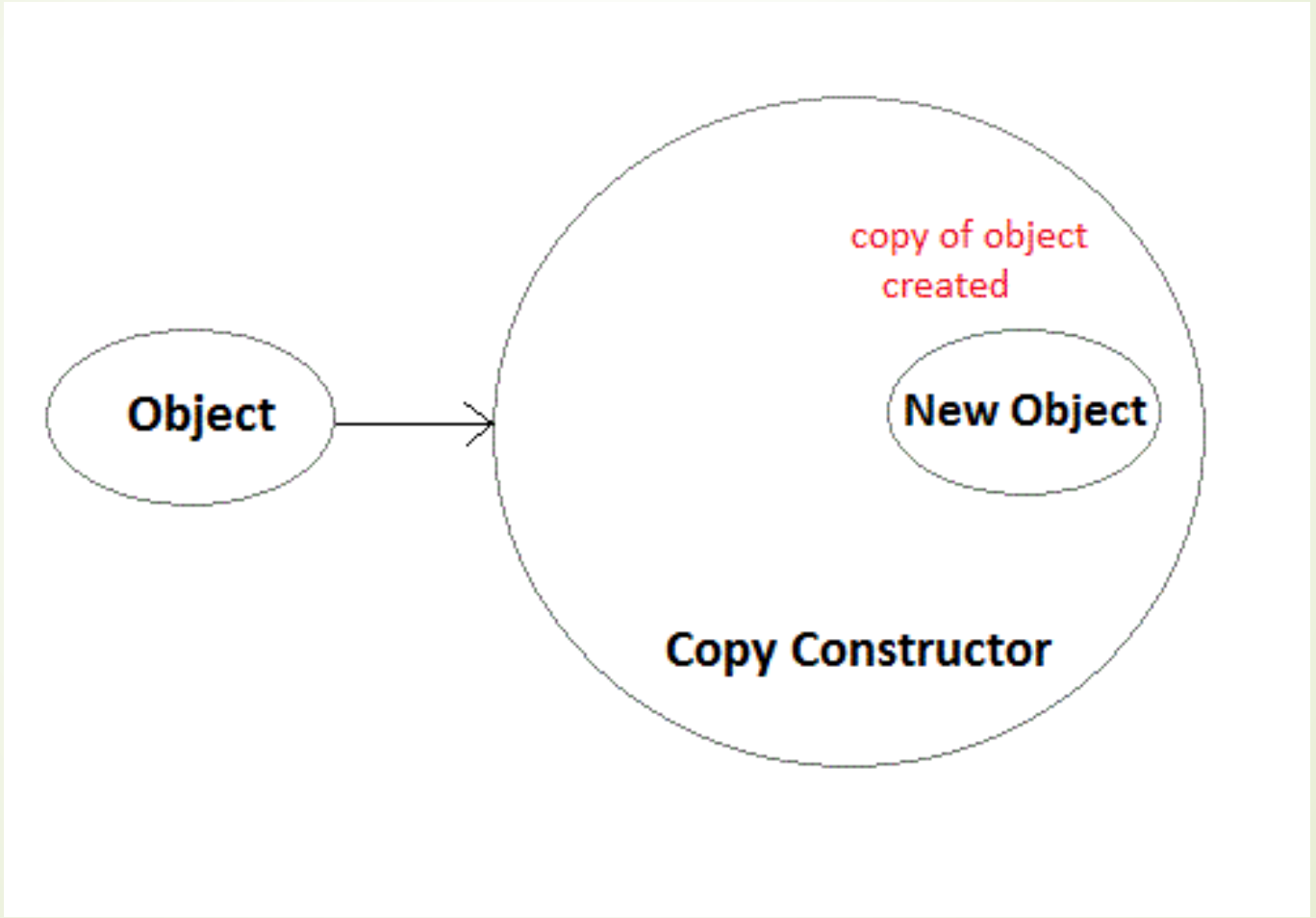
Copy constructor

- A copy constructor is a member function which initializes an object using another object of the same class

ClassName (const ClassName &old_obj);

- The copy constructor is used to −
  - Initialize one object from another of the same type.
  - Copy an object to pass it as an argument to a function.
  - Copy an object to return it from a function.

Copy constructor



Copy constructor

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

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

Why & in Copy Constructor

- So if we pass an argument by value in a copy constructor, a call to copy constructor would be made to call copy constructor which becomes a non-terminating chain of calls. Therefore compiler doesn't allow parameters to be passed by value.

Logically, it should make no sense to modify an object of which you just want to make a copy. That is why we use Const.

Destructors

- We've seen that a special member function—the constructor—is called automatically when an object is first created. You might guess that another function is called automatically when an object is destroyed. This is indeed the case. Such a function is called a *destructor*.
- A destructor has the same name as the constructor (which is the same as the class name) but is preceded by a tilde:

```
class Foo
{
private:
int data;
public:
Foo() : data(0) //constructor (same name as class)
{ }
~Foo() //destructor (same name with tilde)
{ }
};
```

Destructors

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

```
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; }
```

Destructors

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.