COPY CONSTRUCTOR

COPY CONSTRUCTOR

- Although passing simple objects as arguments to functions is a straightforward procedure, some rather unexpected events occur that relate to constructors and destructors.

```
// Constructors, destructors, and passing objects.
#include <iostream>
using namespace std;
class myclass {
                int val;
public:
myclass(int i) { val = i; cout << "Constructing\n"; }
~myclass() { cout << "Destructing\n"; }
 int getval() { return val; }
};
void display(myclass ob)
{
cout << ob.getval() << '\n';
}
int main()
{
myclass a(10);
display(a);
return 0;
}
```

This program produces the following, unexpected output:

```
Constructing
10
Destructing
Destructing
```

When an object is passed to a function, a copy of that object is made (and this copy becomes the parameter in the function). This means that a new object comes into existence. When the function terminates, the copy of the argument (i.e., the parameter) is destroyed. This raises two fundamental questions: First, is the object's constructor called when the copy is made? Second, is the object's destructor called when the copy is destroyed? The answers may, at first, surprise you.

When a copy of an argument is made during a function call, the normal constructor is *not* called. Instead, the object's *copy constructor* is called. A copy constructor defines how a copy of an object is made. However, if a class does not explicitly define a copy constructor, then C++ provides one by default. The default copy constructor creates a bitwise (that is, identical) copy of the object.

However, when the function terminates and the copy of the object used as an argument is destroyed, the destructor *is* called.

A Potential Problem When Passing Objects

If an object used as an argument allocates dynamic memory and frees that memory when it is destroyed, then its local copy inside the function will free the same memory when its destructor is called. This is a problem because the original object is still using the memory. This situation will leave the original object damaged and effectively useless.

```
// Demonstrate a problem when passing
objects.
#include <iostream>
#include <cstdlib>
using namespace std;
class myclass {
int *p;
public:
myclass(int i);
~myclass();
int getval() { return *p; }
};
myclass::myclass(int i)
{
cout << "Allocating p\n";
p = new int;
*p = i;
}
myclass::~myclass()
{
cout << "Freeing p\n";
delete p;
}
```

```
// This will cause a problem.
void display(myclass ob)
{
cout << ob.getval() << '\n';
}
int main()
{
myclass a(10);
display(a);
return 0;
}
```

This program displays the following output:

```
Allocating p
10
Freeing p
Freeing p
```

fundamental error:

When **a** is constructed within **main( )**, memory is allocated and assigned to **a.p**. When **a** is passed to **display( )**, **a** is copied into the parameter **ob**. This means that both **a** and **ob** will have the same value for **p**. That is, both objects will have their copies of **p** pointing to the same dynamically allocated memory. When **display( )** terminates, **ob** is destroyed, and its destructor is called. This causes **ob.p** to be freed. However, the memory freed by **ob.p** is the same memory that is still in use by **a.p**! This is, in itself, a serious bug.

A Potential Problem When Returning Objects

When an object is returned by a function, a temporary object is automatically created, which holds the return value. It is this object that is actually returned by the function. After the value has been returned, this object is destroyed.

```
// An error generated by returning an object.
#include <iostream>
#include <cstring>
#include <cstdlib>
using namespace std;
class sample {
char *s;
public:
sample() { s = 0; }
~sample() { if(s) delete [] s; cout << "Freeing s\n"; }
void show() { cout << s << "\n"; }
void set(char *str);
};

// Load a string.
void sample::set(char *str)
{
s = new char[strlen(str)+1];
strcpy(s, str);
}
// Return an object of type sample.
sample input()
{
char instr[80];
sample str;
cout << "Enter a string: ";
cin >> instr;
str.set(instr);
return str;
}
int main()
{
sample ob;
// assign returned object to ob
ob = input(); // This causes an error!!!!
ob.show(); // displays garbage
return 0;
}
```

The output from this program is shown here:

```
Enter a string: Hello
Freeing s
Freeing s
garbage here
Freeing s
```

The key point to understand from this example is that when an object is returned from a function, the temporary object holding the return value will have its destructor called. Thus, you should avoid returning objects in which this situation can be harmful. One solution is to return either a pointer or a reference. However, this is not always feasible. Another way to solve this problem involves the use of a copy constructor, which is described next.

Creating and Using 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. For example, assuming a class called **myclass**, and **y** as an object of type **myclass**, then the following statements would invoke the **myclass** copy constructor:

```
myclass x = y;          // y explicitly initializing x
func1(y);               // y passed as a parameter
y = func2();            // y receiving a returned object
```

```
// Use a copy constructor to construct a parameter.
#include <iostream>
#include <cstdlib>
using namespace std;
class myclass {
int *p;
public:
myclass(int i); // normal constructor
myclass(const myclass &ob); // copy constructor
~myclass();
int getval() { return *p; }
};
// Copy constructor.
myclass::myclass(const myclass &obj)
{
p = new int;
*p = *obj.p; // copy value
cout << "Copy constructor called.\n";
}
```

This program displays the following output:

```
Allocating p
Copy constructor called.
10
Freeing p
Freeing p
```

```
// Normal Constructor.
myclass::myclass(int i)
{
cout << "Allocating p\n";
p = new int;
*p = i;
}
myclass::~myclass()
{
cout << "Freeing p\n";
delete p;
}
// This function takes one object parameter.
void display(myclass ob)
{
cout << ob.getval() << '\n';
}
int main()
{
myclass a(10);
display(a);
return 0;
}
```

Here is what occurs when the program is run: When **a** is created inside **main( )**, the normal constructor allocates memory and assigns the address of that memory to **a.p**.

Next, **a** is passed to **ob** of **display( )**. When this occurs, the copy constructor is called, and a copy of **a** is created. The copy constructor allocates memory for the copy, and a pointer to that memory is assigned to the copy's **p** member.

Next, the value stored at the original object's **p** is assigned to the memory pointed to by the copy's **p**. Thus, the areas of memory pointed to by **a.p** and **ob.p** are separate and distinct, but the values that they point to are the same. If the copy constructor had not been created, then the default bitwise copy would have caused **a.p** and **ob.p** to point to the same memory.

When **display( )** returns, **ob** goes out of scope. This causes its destructor to be called, which frees the memory pointed to by **ob.p**.

Finally, when **main( )** returns, **a** goes out of scope, causing its destructor to free **a.p**. As you can see, the use of the copy constructor has eliminated the destructive side effects associated with passing an object to a function.

Copy Constructors and Initializations

```
// The copy constructor is called for initialization.
#include <iostream>
#include <cstdlib>
using namespace std;
class myclass {
int *p;
public:
myclass(int i); // normal constructor
myclass(const myclass &ob); // copy constructor
~myclass();
int getval() { return *p; }
};
// Copy constructor.
myclass::myclass(const myclass &ob)
{
p = new int;
*p = *ob.p; // copy value
cout << "Copy constructor allocating p.\n";
}
// Normal constructor.
myclass::myclass(int i)
{
cout << "Normal constructor allocating p.\n";
p = new int;
*p = i;
}
myclass::~myclass()
{
cout << "Freeing p\n";
delete p;
}
```

```
int main()
{
myclass a(10); // calls normal constructor
myclass b = a; // calls copy constructor
return 0;
}
```

This program displays the following output:

Normal constructor allocating p.
Copy constructor allocating p.
Freeing p
Freeing p

Using Copy Constructors When an Object Is Returned

```
/* Copy constructor is called when a temporary object
is created as a function return value.
*/
#include <iostream>
using namespace std;
class myclass {
public:
myclass() { cout << "Normal constructor.\n"; }
myclass(const myclass &obj) { cout << "Copy constructor.\n"; }
};
myclass f()
{
myclass ob; // invoke normal constructor
return ob; // implicitly invoke copy constructor
}
int main()
{
myclass a; // invoke normal constructor
a = f(); // invoke copy constructor
return 0;
}
```

This program displays the following output:

```
Normal constructor.
Normal constructor.
Copy constructor.
```