Function Overloading

Overloading constructor functions

- There are three main reasons why we will want to overload a constructor function:

    -- to gain flexibility

    -- to support array

    -- to create copy constructor

- If a program attempts to create an object for which no matching constructor is found, a compile-time error occurs.

Overloading Constructor Functions

Example:

```
#include <iostream>
using namespace std;

class myclass
{
    int x;
public:
    // overload constructor two ways
    myclass() { x = 0; } // no initializer
    myclass(int n) { x = n; } // initializer
    int getx() { return x; }
};
```

Overloading Constructor Functions

```
int main()
{
    myclass o1(10); // declare with initial value
    myclass o2; // declare without initializer

    cout << "o1: " << o1.getx() << '\n';
    cout << "o2: " << o2.getx() << '\n';

    return 0;
}
```

Overloading Constructor Functions

Another common reason constructor functions are overloaded to allow both individual objects and array of objects to occur within a program.

```
#include <iostream>
using namespace std;

class myclass
{
    int x;
public:
    // overload constructor two ways
    myclass() { x = 0; } // no initializer
    myclass(int n) { x = n; } // initializer
    int getx() { return x; }
};
```

Overloading Constructor Functions

```
int main()
{
    myclass o1[10]; // declare array without initializers

    // declare without initializer
    myclass o2[10] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};

    int i;

    for(i=0; i<10; i++)
    {
        cout << "o1[" << i << "]: " << o1[i].getx();
        cout << '\n';
        cout << "o2[" << i << "]: " << o2[i].getx();
        cout << '\n';
    }

    return 0;
}
```

Creating and Using a Copy Constructor

Preceding chapter shown, problems can occur when an object is passed to or returned from a function.

- One way to avoid these problems to define a **copy constructor**.

Creating and Using a Copy Constructor

- It is important to understand that C++ defines two distinct types of situations in which the value of one object is given to another.
- The first situation is **assignment**.
- The second situation is **initialization** which can **occur three ways**:

1. when an object is used to initialize another in a declaration statement.

2. when an object is passed as a parameter to a function, and

3. a temporary object is created for use as a return value of a function.

Creating and Using a Copy Constructor

- The copy constructor only applies to initializations.
- It does not apply to assignment.
- By default, when an initialization occurs, the compiler will automatically provide a **bitwise copy**.
- However, it is possible to specify precisely how one object will initialize another by defining a copy constructor.
- Once defined, the copy constructor is called whenever an object is used to initialize another.

Creating and Using a Copy Constructor

- The most common form of copy constructor is shown here:

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

Creating and Using a Copy Constructor

- **For example,** assuming a class called myclass, and **y** is an object of type **myclass**, the following statements would invoke the **myclass** copy constructor.

```
myclass x = y; // y explicitly initializing x
fun1(y);       // y passed as a parameter
y = func2();   // y receiving a returned object
```

- In the first two cases, a reference to **y** would be passed to a copy constructor.
- In the third, a reference to the **object returned** by **func( )** is passed to the copy constructor.

Creating and Using a Copy Constructor

- **Here is an example** that illustrates why an explicit copy constructor function is needed.

```
/*
   This program creates a "safe" array class. Since space
   for the array is dynamically allocated, a copy constructor
    is provided to allocate memory when one array object is
    used to initialize another.
*/
#include <iostream>
#include <cstdlib>
using namespace std;
```

```
class array
{
    int *p;
    int size;
public:
    array(int sz) // constructor
    {
        p = new int[sz];
        if(!p)
            exit(1);
        size = sz;
        cout << "Using 'normal' constructor\n";
    }
    ~array() { delete [] p; }

    //copy constructor
    array(const array &a);

    void put(int i, int j)
    {
        if(i>=0 && i<size)
            p[i] = j;
    }
    int get(int i)
    {
        return p[i];
    }
}
```

```
/*
   Copy constructor

   In the following, memory is allocated specifically
   for the copy, and the address of this memory is assigned
    to p. Therefore, p is not pointing to the same
    dynamically allocated memory as the original object.
*/
array::array(const array &a)
{
    int i;

    size = a.size;
    p = new int[a.size]; // allocate memory for copy
    if(!p)
        exit(1);
    for(i=0; i<a.size; i++)
        p[i] = a.p[i]; // copy contents
    cout << "Using copy constructor\n";
}
```

```
int main()
{
    array num(10); // this calls "normal" constructor
    int i;

    // put some values into the array
    for(i=0; i<10; i++)
        num.put(i, i);

    // display num
    for(i=9; i>=0; i--)
        cout << num.get(i);
    cout << "\n";

    // create another array and initialize with num
    array x = num; // this invokes copy constructor

    // display x
    for(i=0; i<10; i++)
        cout << x.get(i);

    return 0;
}
```

When **num** is used to initialize **x**, the copy constructor is called, memory for the new array is allocated and stored in **x.p**, and the contents of **num** are copied to **x**'s array. In this way, **x** and **num** have arrays that have the same values, but each array is separate and distinct. (That is, **num.p** and **x.p** do not point to the same piece of memory.) If the copy constructor had not been created, the bitwise initialization **array x = num** would have resulted in **x** and **num** sharing the same memory for their arrays! (That is, **num.p** and **x.p** would have, indeed, pointed to the same location.)

Creating and Using a Copy Constructor

- The copy constructor is called only for initialization.
- For example, the following sequence does not call the copy constructor defined in the preceding program:

```
array a(10);
array b(10);

b = a; // does not call copy constructor
```

In this case, **b = a** performs the assignment operation.

Creating and Using a Copy Constructor

- To see how the copy constructor helps prevent some of the problems associated with passing certain types of objects to functions, consider the following **(incorrect**) program.

```
// This program has an error.
#include <iostream>
#include <cstring>
#include <cstdlib>
using namespace std;

class strtype
{
    char *p;
public:
    strtype(char *s);
    ~strtype() { delete [] p; }
    char *get() { return p; }
};
```

Creating and Using a Copy Constructor

```
strtype::strtype(char *s)
{
    int l;

    l = strlen(s)+1;

    p = new char [l];
    if(!p)
    {
        cout << "Allocation error\n";
        exit(1);
    }

    strcpy(p, s);
}

void show(strtype x)
{
    char *s;

    s = x.get();
    cout << s << "\n";
}
```

Creating and Using a Copy Constructor

```
int main()
{
    strtype a("Hello"), b("There");

    show(a);
    show(b);

    return 0;
}
```

In this program, when a **strtype** object is passed to **show()**, a bitwise copy is made (since no copy constructor has been defined) and put into parameter **x**. Thus, when the function returns, **x** goes out of scope and is destroyed. This, of course, causes **x**'s destructor to be called, which frees **x.p**. However, the memory being freed is the same memory that is still being used by the object used to call the function. This results in an error.

Creating and Using a Copy Constructor

- The solutions to the preceding problem is to define a copy constructor for the **strtype** class that allocate memory for the copy when the copy is created.

```
#include <iostream>
#include <cstring>
#include <cstdlib>
using namespace std;

class strtype
{
    char *p;
public:
    strtype(char *s); // constructor
    strtype(const strtype &o); // copy constructor
    ~strtype() { delete [] p; } // destructor
    char *get() { return p; }
};
```

Creating and Using a Copy Constructor

```
// "Normal" constructor
strtype::strtype(char *s)
{
    int l;

    l = strlen(s)+1;

    p = new char [l];
    if(!p)
    {
        cout << "Allocation error\n";
        exit(1);
    }

    strcpy(p, s);
}
```

Creating and Using a Copy Constructor

```
// Copy constructor
strtype::strtype(const strtype &o)
{
    int l;

    l = strlen(o.p)+1;

    p = new char [l]; // allocate memory for new copy
    if(!p)
    {
        cout << "Allocation error\n";
        exit(1);
    }

    strcpy(p, o.p); // copy string into copy
}
```

Creating and Using a Copy Constructor

```
void show(strtype x)
{
    char *s;

    s = x.get();
    cout << s << "\n";
}

int main()
{
    strtype a("Hello"), b("There");

    show(a);
    show(b);

    return 0;
}
```

Now when **show()** terminates and **x** goes out of scope, the memory pointed to by **x.p** (which will be freed) is not the same as the memory still in use by the object passed to the function.

Using Default Arguments

- It allows you to give a parameter a default value when on corresponding argument is specified when the function is called.
- For example, this function gives its two parameters default values of 0:

```
void f(int a=0, int b=0);
```

- This function can be called three different ways:

```
f(); // a and b default to 0
f(10); // a is 10, b defaults to 0
f(10, 99); // a is 10, b is 99
```

Using Default Arguments

- Example

```
// A simple first example of default arguments.
#include <iostream>
using namespace std;

void f(int a=0, int b=0)
{
    cout << "a: " << a << ", b: " << b;
    cout << '\n';
}

int main()
{
    f();
    f(10);
    f(10, 99);

    return 0;
}
```

Using Default Arguments

- To understand how default arguments are related to function overloading

```
// Compute area of a rectangle using overloaded functions.
#include <iostream>
using namespace std;

// Return area of a non-square rectangle.
double rect_area(double length, double width)
{
    return length * width;
}

// Return area of a square
```

```
double rect_area(double length)
{
    return length * length;
}
```

Using Default Arguments

```
int main()
{
    cout << "10 x 5.8 rectangle has area: ";
    cout << rect_area(10.0, 5.8) << '\n';

    cout << "10 x 10 square has area: ";
    cout << rect_area(10.0) << '\n';

    return 0;
}
```

Using Default Arguments

- It is not only legal to give constructor functions default arguments, it is also common.

```
#include <iostream>
using namespace std;

class myclass
{
    int x;
public:
    /*
       Use default argument instead of overloading
       myclass's constructor.
    */
    myclass(int n = 0) { x = n; }
    int getx() { return x; }
};
```

Using Default Arguments

```
int main()
{
    myclass o1(10); // declare with initial value
    myclass o2; // declare without initializer

    cout << "o1: " << o1.getx() << '\n';
    cout << "o2: " << o2.getx() << '\n';

    return 0;
}
```