PASSING PARAMETERS TO BASE CLASS CONSTRUCTORS

Passing Parameters to Base-Class Constructors

So far, none of the preceding examples have included constructors that require arguments. In cases where only the derived class' constructor requires one or more parameters, you simply use the standard parameterized constructor syntax. However, how do you pass arguments to a constructor in a base class? The answer is to use an expanded form of the derived class's constructor declaration that passes along arguments to one or more base-class constructors. The general form of this expanded derived-class constructor declaration is shown here:

derived-constructor(arg-list) : base1(arg-list),
base2(arg-list),
// ...
baseN(arg-list)
{
// body of derived constructor
}

Here, *base1* through *baseN* are the names of the base classes inherited by the derived class. Notice that a colon separates the derived class' constructor declaration from the base-class specifications, and that the base-class specifications are separated from each other by commas, in the case of multiple base classes. Consider this program:

```
#include <iostream>
using namespace std;
class base {
protected:
int i;
public:
base(int x) { i=x; cout << "Constructing base\n"; }
~base() { cout << "Destructing base\n"; }
};
class derived: public base {
int j;
public:
```

```
// derived uses x; y is passed along to base.
derived(int x, int y): base(y)
{ j=x; cout << "Constructing derived\n"; }
~derived() { cout << "Destructing derived\n"; }
void show() { cout << i << " " << j << "\n"; }
};
int main()
{
derived ob(3, 4);
ob.show(); // displays 4 3
return 0;
}
```

Here, **derived**'s constructor is declared as taking two parameters, **x** and **y**. However, **derived( )** uses only **x; y** is passed along to **base( ).** In general, the derived class' constructor must declare both the parameter(s) that it requires as well as any required by the base class. As the example illustrates, any parameters required by the base class are passed to it in the base class' argument list specified after the colon.

Here is an example that uses multiple base classes:

```
#include <iostream>
using namespace std;
class base1 {
protected:
int i;
public:
base1(int x) { i=x; cout << "Constructing base1\n"; }
~base1() { cout << "Destructing base1\n"; }
};
class base2 {
protected:
int k;
public:
```

```
base2(int x) { k=x; cout << "Constructing base2\n"; }
~base2() { cout << "Destructing base1\n"; }
};
class derived: public base1, public base2 {
int j;
public:
derived(int x, int y, int z): base1(y), base2(z)
{ j=x; cout << "Constructing derived\n"; }
~derived() { cout << "Destructing derived\n"; }
void show() { cout << i << " " << j << " " << k << "\n"; }
};
int main()
{
derived ob(3, 4, 5);
ob.show(); // displays 4 3 5
return 0;
}
```

It is important to understand that arguments to a base-class constructor are passed via arguments to the derived class' constructor. Therefore, even if a derived class' constructor does not use any arguments, it will still need to declare one if the base class requires it. In this situation, the arguments passed to the derived class are simply passed along to the base.

For example, in this program, the derived class' constructor takes no arguments, but **base1( )** and **base2( )** do:

```
#include <iostream>
using namespace std;
class base1 {
protected:
int i;
public:
base1(int x) { i=x; cout << "Constructing base1\n"; }
~base1() { cout << "Destructing base1\n"; }
```

```
};
class base2 {
protected:
int k;
public:
base2(int x) { k=x; cout << "Constructing base2\n"; }
~base2() { cout << "Destructing base2\n"; }
};
class derived: public base1, public base2 {
public:
/* Derived constructor uses no parameter, but still must be declared as taking them to pass them along to base classes. */
derived(int x, int y): base1(x), base2(y)
{ cout << "Constructing derived\n"; }
~derived() { cout << "Destructing derived\n"; }
void show() { cout << i << " " << k << "\n"; }
};
int main()
{
derived ob(3, 4);
ob.show(); // displays 3 4
return 0;
}
```

A derived class' constructor is free to make use of any and all parameters that it is declared as taking, even if one or more are passed along to a base class. Put differently, passing an argument along to a base class does not preclude its use by the derived class as well. For example, this fragment is perfectly valid:

```
class derived: public base {
int j;
public:
// derived uses both x and y and then passes them to base.
```

```
derived(int x, int y): base(x, y)
{ j = x*y; cout << "Constructing derived\n"; }
```

One final point to keep in mind when passing arguments to base-class constructors: The argument can consist of any expression valid at the time. This includes function calls and variables. This is in keeping with the fact that C++ allows dynamic initialization.

Source : http://elearningatria.files.wordpress.com/2013/10/cse-iii-object-oriented-programming-with-c-10cs36-notes.pdf