Chapter 2. Constructors, Destructors, and Assignment Operators

Almost every class you write will have one or more constructors, a destructor, and a copy assignment operator. Little wonder. These are your bread-and-butter functions, the ones that control the fundamental operations of bringing a new object into existence and making sure it's initialized, getting rid of an object and making sure it's properly cleaned up, and giving an object a new value. Making mistakes in these functions will lead to far-reaching — and unpleasant — repercussions throughout your classes, so it's vital that you get them right. In this chapter, I offer guidance on putting together the functions that comprise the backbone of well-formed classes.

PREV < Day Day Up > NEXT

Item 5: Know what functions C++ silently writes and calls

When is an empty class not an empty class? When C++ gets through with it. If you don't declare them yourself, compilers will declare their own versions of a copy constructor, a copy assignment operator, and a destructor. Furthermore, if you declare no constructors at all, compilers will also declare a default constructor for you. All these functions will be both `public` and `inline` (see Item 30). As a result, if you write

```
class Empty{};
```

it's essentially the same as if you'd written this:

```
class Empty {
public:
  Empty() { ... }                               // default constructor
  Empty(const Empty& rhs) { ... }               // copy constructor

  ~Empty() { ... }                              // destructor — see below
                                                // for whether it's virtual

  Empty& operator=(const Empty& rhs) { ... } // copy assignment operator
};
```

These functions are generated only if they are needed, but it doesn't take much to need them. The following code will cause each function to be generated:

```
Empty e1;                                  // default constructor;
                                           // destructor

Empty e2(e1);                              // copy constructor

e2 = e1;                                   // copy assignment operator
```

Given that compilers are writing functions for you, what do the functions do? Well, the default constructor and the destructor primarily give compilers a place to put "behind the scenes" code such as invocation of constructors and destructors of base classes and non-static data members. Note that the generated destructor is non-virtual (see Item7) unless it's for a class inheriting from a base class that itself declares a virtual destructor (in which case the function's virtualness comes from the base class).

As for the copy constructor and the copy assignment operator, the compiler-generated versions simply copy each non-static data member of the source object over to the target object. For example, consider a `NamedObject` template that allows you to associate names with objects of type `T`:

```
template<typename T>

class NamedObject {

public:

  NamedObject(const char *name, const T& value);

  NamedObject(const std::string& name, const T& value);


  ...


private:

  std::string nameValue;

  T objectValue;

};
```

Because a constructor is declared in `NamedObject`, compilers won't generate a default constructor. This is important. It means that if you've carefully engineered a class to require constructor arguments, you don't have to worry about compilers overriding your decision by blithely adding a constructor that takes no arguments.

`NamedObject` declares neither copy constructor nor copy assignment operator, so compilers will generate those functions (if they are needed). Look, then, at this use of the copy constructor:

```
NamedObject<int> no1("Smallest Prime Number", 2);


NamedObject<int> no2(no1);                  // calls copy constructor
```

The copy constructor generated by compilers must initialize `no2.nameValue` and `no2.objectValue` using `no1.nameValue` and `no1.objectValue`, respectively. The type of `nameValue` is `string`, and the standard `string` type has a copy constructor, so `no2.nameValue` will be initialized by calling the `string` copy constructor with `no1.nameValue` as its argument. On the other hand, the type of `NamedObject<int>::objectValue` is `int` (because `T` is `int` for this template instantiation), and `int` is a built-in type, so `no2.objectValue` will be initialized by copying the bits in `no1.objectValue`.

The compiler-generated copy assignment operator for `NamedObject<int>` would behave essentially the same way, but in general, compiler-generated copy assignment operators behave as I've described only when the resulting code is both legal and has a reasonable chance of making sense. If either of these tests fails, compilers will refuse to generate an `operator=` for your class.

For example, suppose `NamedObject` were defined like this, where `nameValue` is a *reference* to a string and `objectValue` is a *const* `T`:

```
template<class T>
```

```
class NamedObject {
public:
  // this ctor no longer takes a const name, because nameValue
  // is now a reference-to-non-const string. The char* constructor
  // is gone, because we must have a string to refer to.
  NamedObject(std::string& name, const T& value);

  ...                                // as above, assume no
                                     // operator= is declared
private:
  std::string& nameValue;            // this is now a reference
  const T objectValue;               // this is now const
};
```

Now consider what should happen here:

```
std::string newDog("Persephone");
std::string oldDog("Satch");

NamedObject<int> p(newDog, 2);              // when I originally wrote this, our
                                            // dog Persephone was about to
                                            // have her second birthday

NamedObject<int> s(oldDog, 36);             // the family dog Satch (from my
                                            // childhood) would be 36 if she
                                            // were still alive

p = s;                                      // what should happen to
                                            // the data members in p?
```

Before the assignment, both `p.nameValue` and `s.nameValue` refer to `string` objects, though not the same ones. How should the assignment affect `p.nameValue`? After the assignment, should `p.nameValue` refer to the `string` referred to by `s.nameValue`, i.e., should the reference itself be modified? If so, that breaks new ground, because C++ doesn't provide a way to make a reference refer to a different object. Alternatively, should the `string` object to which `p.nameValue` refers be modified, thus affecting other objects that hold pointers or references to that `string`, i.e.,

objects not directly involved in the assignment? Is that what the compiler-generated copy assignment operator should do?

Faced with this conundrum, C++ refuses to compile the code. If you want to support assignment in a class containing a reference member, you must define the copy assignment operator yourself. Compilers behave similarly for classes containing `const` members (such as `objectValue` in the modified class above). It's not legal to modify `const` members, so compilers are unsure how to treat them during an implicitly generated assignment function. Finally, compilers reject implicit copy assignment operators in derived classes that inherit from base classes declaring the copy assignment operator `private`. After all, compiler-generated copy assignment operators for derived classes are supposed to handle base class parts, too (see Item 12), but in doing so, they certainly can't invoke member functions the derived class has no right to call.

Things to Remember

- Compilers may implicitly generate a class's default constructor, copy constructor, copy assignment operator, and destructor.

PREV < Day Day Up > NEXT