This is a pre-publication draft of the column I wrote for the June 1995 issue of the *C++ Report*. "Pre-publication" means this is what I sent to the *Report*, but it may not be exactly the same as what appeared in print, because the *Report* and I often make small changes after I submit the "final" draft for a column. Comments? Feel free to send me mail: `smeyers@aristeia.com`.

Bounding Object Instantiations, Part 2

In my last column (March-April 1995), I introduced the problem of bounding the number of instantiations of a class, and I considered how to allow zero and one instances. I also discussed how there's ambiguity surrounding what it means to "instantiate" an object, because objects can exist in three different contexts: on their own, as the base class part of a more derived object, and as a member of an enclosing object. In this column I'll continue the discussion of bounding object instantiations, and I'll bring it to a close.

We now know how to design a class that allows but a single instantiation, we know that keeping track of the number of objects of a particular class is complicated by the fact that object constructors are called in three different contexts, and we know that we can eliminate the confusion surrounding object counts by making constructors private. It is time, therefore, to revisit our original goal, that of limiting the number of objects of a particular type.

Before doing that, however, it is worthwhile to make one final observation. Our use of the `thePrinter` function to encapsulate access to a single object certainly limits the number of `Printer` objects to one, but it also limits us to a single `Printer` object for each program run. As a result, it's not possible to write code like this:

```
create Printer object p1;

use p1;

destroy p1;

create Printer object p2;

use p2;

destroy p2;

...
```

This design never instantiates more than a single `Printer` object at a time, but it does use different `Printer` objects in different parts of the program. It somehow seems unreasonable that this isn't allowed. After all, at no point do we violate the constraint that only one printer may exist. Isn't there a way to make this legal while still honoring the restriction that we can't have more than a single printer?

There is. All we have to do is combine the object-counting code we used earlier with the pseudo-constructors we saw near the end of my last column:

```
class Printer {
private:
  static unsigned short numObjects = 0;

  Printer();
  Printer(const Printer& rhs);

public:
  class TooManyObjects{};

public:
  // pseudo-constructors
  static Printer * makePrinter();
  static Printer * makePrinter(const Printer& rhs);

  ~Printer();

  void submitJob(const PrintJob& job);
  void reset();
  void performSelfTest();
  ...

};

// Obligatory definitions of class static
unsigned short Printer::numObjects;

Printer::Printer()
{
  if (numObjects >= 1) {
    throw TooManyObjects();
  }

  proceed with normal object construction here;

  ++numObjects;
}

Printer::Printer(const Printer& rhs)
{
  if (numObjects >= 1) {
    throw TooManyObjects();
  }

  proceed with normal object construction here;

  ++numObjects;
}

Printer * Printer::makePrinter()
{ return new Printer; }
```

```
Printer * Printer::makePrinter( const
                               Printer& rhs)
{ return new Printer(rhs); }
```

If the notion of throwing an exception when too many objects are requested strikes you as unreasonably harsh, you could have the pseudo-constructors return 0 instead. Clients would then have to check for the null pointer before doing anything with it, of course.

Clients use this `Printer` class just like they would any other class, except they must call pseudo-constructor functions instead of regular constructors and they must delete the objects they create:

```
Printer p1;             // error! default ctor is
                        // private

Printer *p2 =
  Printer::makePrinter();
                        // fine, indirectly calls
                        // default ctor

Printer p3 = *p2;       // error! copy ctor is
                        // private

Printer *p4 =
  Printer::makePrinter(p3);
                        // fine, indirectly calls
                        // copy ctor (but throws
                        // an exception because
                        // *p2 already exists)

p2->performSelfTest();  // all other functions
p2->reset();            // are called as usual

...

delete p2;              // avoid memory leak
```

This technique is trivially generalizable to any number of objects. All we have to do is replace the hard-wired constant 1 in the code above with a class-specific value. For example, the following revised implementation of our `Printer` class allows up to 10 `Printer` objects to exist at any given time:

```
class Printer {
private:
  static unsigned short numObjects = 0;
  static unsigned short maxObjects = 10;

  ...                      // as above

};

// Obligatory definitions of class statics
unsigned short Printer::numObjects;
unsigned short Printer::maxObjects;

Printer::Printer()
{
  if (numObjects >= maxObjects) {
```

```
    throw TooManyObjects();
  }

  ...

}

Printer::Printer(const Printer& rhs)
{
  if (numObjects >= maxObjects) {
    throw TooManyObjects();
  }

  ...

}
```

This approach works like the proverbial charm, but there is one aspect of it that continues to nag. If we had a lot of classes like `Printer` whose instantiations needed to be limited, we'd have to write this same code over and over, once per class. That would be mind-numbingly dull. Given a fancy-pants language like C++, it somehow seems that we should be able to automate the process. Isn't there a way to encapsulate the notion of counting instances and bundle it into a class?

Surely we can come up with a base class for counting object instances and have classes like `Printer` inherit from that, but it turns out we can do even better. We can actually come up with a way to encapsulate the whole counting kit and kaboodle, by which I mean not only the functions to manipulate the instance count, but also the instance count itself.

The counter in the `Printer` class is the static variable `numObjects`, so we need to move that variable into an instance-counting class. However, we also need to make sure that each class for which we're counting instances has a *separate* counter. Use of a counting class *template* lets us automatically generate the appropriate number of counters with the greatest of ease, because we can make the counter a static member of the classes generated from the template:

```
template<class BeingCounted>
class Counted {
private:
  static unsigned short numObjects = 0;
  static unsigned short maxObjects;

protected:
  Counted();
  Counted(const Counted& rhs);

  ~Counted() { --numObjects; }

public:
  class TooManyObjects{};     // for throwing exceptions

  static int objectCount() { return numObjects; }
};
```

```
template<class BeingCounted>
Counted::Counted()
{
  if (numObjects >= maxObjects) throw TooManyObjects();
  ++numObjects;
}

template<class BeingCounted>
Counted::Counted(const Counted&)
{
  if (numObjects >= maxObjects) throw TooManyObjects();
  ++numObjects;
}
```

The classes generated from this template are designed only to be used as base classes, hence the protected constructors and destructor.

You may notice that the `objectCount` function has a return type of `int`, but it actually returns a value that is `unsigned`. This seems like a loss-of-precision error just waiting to happen, and some compilers get so nervous about it they issue a warning. Unfortunately, returning the `unsigned short` directly isn't much better (most callers will just treat it as an `int` anyway), so we'll just hope an `int` is bigger than a `short` or we never have more objects than an `int` can represent. This is hardly iron-clad software engineering, but the alternative — eschewing unsigned types entirely — isn't that appealing, either.

We can now modify the `Printer` class to use the `Counted` template:

```
class Printer: private Counted<Printer> {
private:
  Printer();
  Printer(const Printer& rhs);

public:
  // pseudo-constructors
  static Printer * makePrinter();
  static Printer * makeprinter( const
                                Printer& rhs);

  ~Printer();

  void submitJob(const PrintJob& job);
  void reset();
  void performSelfTest();
  ...

  using Counted<Printer>::objectCount;
};
```

Now, the fact that `Printer` uses the `Counted` template to keep track of how many `Printer` objects exist is, frankly, nobody's business but the author of `Printer`'s. Such implementation details are best kept private, and that's why private inheritance is used here.

Quite properly, most of what `Counted` does is hidden from `Printer`'s clients, but those clients might reasonably want to find out how many Printer objects exist. The Counted template provides the objectCount function to provide this information, but that function becomes private in Printer due to our use of private inheritance. To restore the public accessibility of that function, we employ a using declaration:

```
class Printer: private Counted<Printer> {
...

public:
  using Counted<Printer>::objectCount;
                                    // make this function
                                    // public for clients
  ...                               // of Printer
};
```

This is perfectly legitimate, according to the emerging language standard, but if your compilers don't yet support namespaces, they are unlikely to allow it. If they don't, you can use the older access-declaration syntax:

```
class Printer: private Counted<Printer> {
...

public:
  Counted<Printer>::objectCount;
                             // make objectCount
  ...                        // public in Printer
};
```

This more traditional syntax has the same meaning as the using declaration, but it's *deprecated*, meaning it will eventually be removed from the language. Needless to say, you should avoid the use of deprecated features whenever you can, but don't lose too much sleep if you find that your compilers force you into a little deprecation now and then. It will be some time — years, probably — before such deprecated features are actually made illegal.

When Printer inherits from Counted<Printer>, it can completely forget about counting objects. The class can be written as if somebody else were doing the counting for it, because somebody else (Counted<Printer>) is. For example a Printer constructor now looks like this:

```
Printer::Printer()
{
  proceed with normal object construction;
}
```

What's interesting here is not what you see, it's what you don't. No checking of the number of objects to see if the limit is about to be exceeded, no incrementing of the number of objects in existence once the constructor is done. All that is now handled by the Counted<Printer> constructors, and because Counted<Printer> is a base class of Printer, we know that a Counted<Printer> constructor will always be called before a Printer constructor is. If too many objects are created, a

Counted<Printer> constructor will throw an exception, and the Printer constructor won't even be invoked. Nifty, huh?

Nifty or not, however, there's one loose end that demands to be tied, and that's the mandatory definitions of the statics inside Counted. It's easy enough to take care of numObjects — we just put this in Counted's implementation file:

```
template<class BeingCounted>
unsigned short Counted::numObjects;
```

The situation with maxObjects is a bit trickier. To what value should we initialize this variable? If we want to allow up to 10 printers, we should initialize Counter<Printer>::maxObjects to 10. If, on the other hand, we want to allow up to 16 file descriptor objects, we should initialize Counted<FileDescriptor>::maxObjects to 16. What to do?

We take the easy way out: we do nothing. We provide no initialization at all for maxObjects. Instead, we require that *clients* of the class provide the appropriate initialization. The author of Printer, then, must add this to an implementation file:

```
unsigned short
Counted<Printer>::maxObjects = 10;
```

Similarly, the author of FileDescriptor must add this:

```
unsigned short
Counted<FileDescriptor>::maxObjects = 16;
```

What will happen, you might wonder, if these authors forget to provide a suitable definition for maxObjects? Simple: they'll get an error at link time, because maxObjects will be undefined. Provided we've adequately documented this requirement for clients of Counted, they can then mutter "Duh" to themselves and go back and add the requisite initialization.

Acknowledgment

My design of a template class for counting objects is based on a posting to comp.lang.c++ by Jamshid Afshar.