This is a pre-publication draft of the column I wrote for the March-April 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 1

Okay, you're crazy about objects, but sometimes you'd like to bound your insanity. For example, you've only got one printer in your system, so you'd like to somehow limit the number of printer objects to one. Or you've only got 16 file descriptors you can hand out, so you've got to make sure there are never more than that many file descriptor objects in existence. How can you do such things? How can you limit the number of objects?

Well, if this were a proof by mathematical induction, we'd start with the base case of *n = 1*, then build from there. Fortunately, this is neither a proof nor an induction. Furthermore, it turns out to be more instructive to begin with *n = 0*, so we'll start there instead. How, then, do you prevent objects from being instantiated at all?

Each time an object is instantiated, we know one thing for sure: a constructor will be called. That being the case, the easiest way to prevent objects of a particular class from being created is to declare the constructors of that class *private*:

```
class CantBeInstantiated {
private:
  CantBeInstantiated();
  CantBeInstantiated(const CantBeInstantiated&);

  ...

};
```

When employing this trick, it's often important to declare a default constructor, because if you declare no constructors at all, your ever-helpful compilers will silently generate a public default constructor for you. On the other hand, there's really no need to declare the copy constructor private, because you certainly don't have to worry about keeping objects from being copied if you can't create them in the first place. Still, it's often good practice to emphasize that no constructor is supposed to be accessible, and that's why I've declared the copy constructor private in `CantBeInstantiated` above.

An uninstantiable class is generally pretty useless, but now that we know how to remove everybody's right to create objects, we can selectively loosen the restriction. If, for example, we want to create a class for printers, but we also want to accurately reflect the real-

world fact that there is only one printer available to us, here's a way to encapsulate the printer object inside a function such that everybody has access to the printer, but only a single printer object is ever created:

```
class Printer {
friend Printer& thePrinter();

private:
  Printer();
  Printer(const Printer& rhs);
  ...

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

};

Printer& thePrinter()
{
  static Printer p;    // the one printer object
  return p;
}
```

There are three separate components to this design. First, the constructors of the `Printer` class are private. That suppresses object creation. Second, the global function `thePrinter` is declared a friend of the class. That lets `thePrinter` escape the restriction imposed by the private constructors. Finally, `thePrinter` contains a static `Printer` object. That means only a single object will ever be created. It also means that if `thePrinter` is never called, we won't waste the time and energy to construct the `Printer` object inside it.

Client code just refers to `thePrinter` whenever it wishes to interact with the system's lone printer. By returning a reference to a `Printer` object, `thePrinter` can be used in any context where a `Printer` object itself could be:

```
class PrintJob {
public:
  PrintJob(const char *whatToPrint);

  ...

};

ostrstream buffer;

...                       // write stuff to
                          // buffer
thePrinter().reset();

char *bufContents =
  buffer.str();           // get char* version
                          // of buffer
```

```
thePrinter().submitJob(bufContents);
delete [] bufContents;    // avoid memory leak
```

In this example, we make use of an `ostrstream` object as an unbounded in-memory buffer. We write to it and write to it and write to it until we need write no more, and the `ostrstream` object itself is responsible for all the grungy memory management that must take place to ensure the buffer expands to hold all the data we put into it. Because they allow you to construct strings through sequences of standard output operations, `ostrstream`s are wonderful for tasks like generating identifier names, file pathnames, and error messages on the fly; they're one of the most convenient things in the iostream library. If you're not familiar with `ostrstream`s, dash to the nearest book that describes them (any self-respecting introductory or reference book on C++ will do so) and read yourself silly. You'll be glad you did.

It's possible, of course, that `thePrinter` strikes you as a needless addition to the global namespace. "Yes," you may say, "as a global function it looks more like a global variable, but global variables are gauche, and I'd prefer to localize all printer-related functionality inside the `Printer` class." Well, far be it from me to argue with someone who uses words like gauche. `thePrinter` can just as easily be made a static member function of `Printer`, and that puts it right where you want it. It also eliminates the need for a `friend` declaration, which many regard as tacky in its own right. Using a static member function, `Printer` looks like this:

```
class Printer {
private:
  Printer();
  Printer(const Printer& rhs);
  ...

public:
  static Printer& thePrinter();
  ...

};

Printer& Printer::thePrinter()
{
  static Printer p;
  return p;
}
```

Now, however, clients have to be a bit wordier when they refer to the printer:

```
Printer::thePrinter().reset();

char *bufContents = buffer.str();
Printer::thePrinter().submitJob(bufContents);
delete [] bufContents;
```

Needless to say, this is hardly crippling.

There are two subtleties here worth exploring. First, it's important that the single `Printer` object be static in a *function* and not in a

class. An object that's static in a class is, for all intents and purposes, *always* constructed (and destructed), even if it's never used. In contrast, an object that's static in a function is created the first time through the function, so if the function's never called, the object is never created. One of the philosophical pillars on which C++ was built is the idea that you shouldn't pay for things you don't use, and defining an object like our printer as a static object in a function is one way of adhering to this philosophy.

There is another drawback to making the printer a class static versus a function static, and that has to do with its time of initialization. We know exactly when a function static is initialized: it's the first time through the function at the point where the static is defined. The situation with a class static (or, for that matter, a global static, should you be so gauche as to use one) is less well defined. The emerging C++ standard offers certain guarantees regarding the order of initialization of statics within a particular translation unit (i.e., a body of source code that yields a single object file), but it has *nothing* to say about the initialization order of static objects in different translation units. In practice, this turns out to be a source of countless headaches. Function statics, when they can be made to suffice, avoid these headaches. In our example here, they can, so why suffer?

The second subtlety has to do with the interaction of inlining and static objects inside functions. Look again at the code for `thePrinter`:

```
Printer& Printer::thePrinter()
{
  static Printer p;
  return p;
}
```

Except for the first time through this function (when `p` must be constructed), this is a one-line function — it consists entirely of the statement "`return p;`". If ever there were a good candidate for inlining, this function would certainly seem to be the one. Yet it's not declared `inline`. Why not?

Consider for a moment why you'd declare an object to be static. It's usually because you only want a single copy of that object, right? Now consider what `inline` means. Conceptually, of course, it means the compiler should replace each call to the function with a copy of the function body, but it means something else to your compilers, too. It also means the function in question has *internal linkage*.

I hate to do this to you, but we need to take a brief terminology time-out. If you have a weak heart, you should probably grab your nitroglycerin and sit down before reading the next few paragraphs.

A function with internal linkage (such as an inline function) isn't visible outside the current translation unit; it's treated just like a function in C (or C++) that's declared `static` at file scope. Non-inline functions, on the other hand, have *external linkage* and are visible everywhere in a program, even outside the current translation unit. In other words, compilers treat such functions as if they were explicitly declared `extern`.

Enter the word "static." If you like overloading, you must love this word. When applied to a variable, object, or function at *file scope*, the keyword `static` means "a separate copy for this translation unit." (At file scope, then, static non-inline functions have internal linkage.) When applied to a variable or object at *class or function scope*, `static` means "only one copy of this thing." (When applied to a *function* at class scope, it means "can only access static members of this class," but that meaning doesn't concern us here.)

But wait. Static variables and objects and *all* variables and objects at file, namespace, or global scope are said to have storage class "static." Such variables and objects are initialized once per program run through what is called *static initialization*. Static initialization typically (though not necessarily) occurs prior to calling `main`. Unfortunately, variables and objects that are initialized through static initialization need not have been declared `static`. For example, global variables aren't declared `static`, but they are initialized through a program's static initialization.

By now you must be wondering why on earth you should care. You should care because it affects the behavior of your programs. In particular, the meaning of the function `thePrinter`, if it were to be defined as follows, is almost certainly not what you expect:

```
inline Printer& Printer::thePrinter()
{
  static Printer p;
  return p;
}
```

As an inline function, `thePrinter` has internal linkage, so it's not visible outside a translation unit. You therefore get a *different copy* of `thePrinter` inside each translation unit that uses it. Most programs are made up of many translation units (i.e., object files), so most programs using `thePrinter` would end up with many copies of this function. What, then, of the static object `p` inside `thePrinter`? Well, each translation unit contains a different copy of this function, so each copy of `thePrinter` gets its own copy of `p`! In short, if you put a static object inside an inline function, you end up with *one object per translation unit* that uses the function, not one object for the entire program.

It is highly unlikely that this is what you want.

In fact, it is so unlikely that some compiler vendors ignore the dictates of the nascent standard and give you only a single copy of a function static, even if the function is declared `inline`. Really. Other vendors do what the standard says they're supposed to do. And some offer both behaviors, letting you choose which one you get through a command line switch or some other configuration option.

In view of this state of affairs, inline functions containing static objects are, for all practical purposes, unportable, even if you understand their highly unintuitive semantics. We are only interested in portable programs, so `thePrinter` above is not declared inline.

But, alas, you are stubborn. If, in spite of the foregoing discussion, you are *absolutely positively* determined to inline `thePrinter`, do it this way:

```
extern inline Printer& Printer::thePrinter()
{
  static Printer p;
  return p;
}
```

Here `inline` tells your compilers to give `thePrinter` internal linkage, but `extern` tells them to give it external linkage. It turns out that `extern` trumps `inline` for purposes of linkage specifications, so this should yield a single externally linked inline function with a single static `Printer` object inside it. I say "should" because it would be far from surprising if different compilers did different things, the would-be language standard notwithstanding. It would also be far from surprising if people who read this code have no idea what you are trying to do.

An alternative approach to allowing the creation of only a single object may be found in the new book on design patterns by Gamma, Helm, Johnson, and Vlissides [cite it here]. Their *Singleton* pattern is similar to the approach I just presented, but they avoid the nettlesome issue of function statics by using a static pointer in the *class*. As a result, however, their `thePrinter`-like function must return a pointer, and that affects the syntax that clients must employ.

But maybe you think this business of creating a function to return a reference (or, if you're partial to Gamma *et al.*'s *Singleton* pattern, a pointer) to a hidden object is the wrong way to go about limiting the number of objects in the first place. Perhaps you think it's much more straightforward to simply count the number of objects in existence and throw an exception in a constructor if too many objects are requested. In other words, maybe you think we should handle printer creation like this:

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

  Printer(const Printer& rhs);
                    // there is a limit of 1
                    // printer, so never allow
                    // copying
public:
  class TooManyObjects{};
                    // exception class for use
                    // when too many objects
                    // are requested
public:
  Printer();
  ~Printer();

  ...

};
```

The idea is to use `numObjects` to keep track of how many `Printer` objects are currently in existence. This value will be incremented in the class constructors and decremented in its destructor.

If an attempt is made to construct too many `Printer` objects, we throw an exception of type `TooManyObjects`:

```
// Obligatory definition of the class static
unsigned short Printer::numObjects;

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

  proceed with normal construction here;

  ++numObjects;
}

Printer::~Printer()
{
  perform normal destruction here;

  --numObjects;
}
```

By the way, don't be surprised if your compilers get all upset about the declaration of `Printer::numObjects` in the class definition above. In particular, be prepared for them to complain about the specification of 0 as an initial value for that variable. The ability to specify such initial values inside a class definition was added to C++ only recently, so many compilers don't yet allow it. If your compilers are as-yet-unupdated, send their vendors hate mail, then pacify the compilers by declaring `numObjects` without an initial value:

```
class Printer {
private:
  static unsigned short numObjects;
                              // no initial value
  ...                         // specified
};
```

This has the same effect as the code above, because class statics are implicitly initialized to 0, but explicitly specifying the initial value is a lot easier for other programmers to understand. Furthermore, you can specify any (constant) initial value in the class definition; you're not limited to the value 0, as you are with implicit initialization. When your compilers support the specification of initial values in class definitions, then, you should surely take advantage of that capability.

Initializations of statics aside, this approach to limiting object creation is attractive for a couple of reasons. For one thing, it's very straightforward — everybody should be able to understand what's going on. For another, it's easy to generalize this approach so that the maximum number of allowed objects is some number other than 1.

Unfortunately, there is also a problem with this strategy. Suppose we have a special kind of printer, say, a color printer. The class for

such printers would have much in common with our generic printer class, so of course we'd inherit from it:

```
class ColorPrinter: public Printer {
  ...
};
```

Now suppose we have one generic printer and one color printer in our system:

```
Printer p;
ColorPrinter cp;
```

How many `Printer` objects result from these object definitions? The answer is two: one for `p` and one for the `Printer` part of `cp`. At runtime, then, a `TooManyObjects` exception will be thrown during the construction of the base class part of `cp`. For many programmers, this is neither what they want nor what they expect. (Designs that avoid having concrete classes inherit from other concrete classes do not suffer from this problem. For details on this design philosophy, consult my columns in the July-August 1994, November-December 1994, and January 1995 *C++ Report*s.)

A similar problem occurs when `Printer` objects are contained inside other objects:

```
class CPFMachine {  // for machines that can
private:            // copy, print, and fax

  Printer p;        // for printing capabilities
  FaxMachine f;     // for faxing capabilities
  CopyMachine c;    // for copying capabilities

  ...

};

CPFMachine m1;      // fine

CPFMachine m2;      // throws TooManyObjects
                    // exception
```

The problem is that `Printer` objects can exist in three different contexts: on their own, as base class parts of more derived objects, and embedded inside larger objects. The presence of these different contexts significantly muddies the waters regarding what it means to keep track of the "number of objects in existence," because what you consider to be the existence of an object may not jibe with your compilers'.

Often, you will be interested only in allowing objects to exist on their own, and you will wish to limit the number of *those* kinds of instantiations. That restriction is easy to satisfy if you adopt the strategy exemplified by our original `Printer` class, because the `Printer` constructors are private, and (in the absence of `friend` declarations) classes with private constructors can't be used as base classes, nor can they be embedded inside other objects.

The fact that you can't derive from classes with private constructors leads to a general scheme for preventing derivation, one that

doesn't necessarily have to be coupled with limiting object instantiations. Suppose, for example, you have a class, UPNumber, for representing numbers with unlimited precision. Further suppose you'd like to allow any number of UPNumber objects to be created, but you'd also like to ensure that no class ever inherits from UPNumber. (One reason for doing this might be to justify the presence of a nonvirtual destructor in UPNumber. Classes without virtual functions yield smaller objects than do equivalent classes with virtual functions, because there is no need for each object to carry around a pointer to the class's virtual table.) Here's how you can design UPNumber to satisfy both criteria:

```
class UPNumber {
private:
  UPNumber();
  UPNumber(int initValue);
  UPNumber(double initValue);
  UPNumber(const UPNumber& rhs);
  ...

public:
  // pseudo-constructors
  static UPNumber makeNumber();
  static UPNumber makeNumber(int initValue);
  static UPNumber makeNumber(double initValue);
  static UPNumber makeNumber(  const
                               UPNumber& rhs);
  ...

};

UPNumber UPNumber::makeNumber()
{ return UPNumber(); }

UPNumber UPNumber::makeNumber(int initValue)
{ return UPNumber(initValue); }

UPNumber UPNumber::makeNumber(double initValue)
{ return UPNumber(initValue); }

UPNumber UPNumber::makeNumber(  const
                                UPNumber& rhs)
{ return UPNumber(rhs); }
```

Unlike the thePrinter function that always returned a reference to a single object, each makeNumber pseudo-constructor returns a unique object. That's what allows an unlimited number of UPNumber objects to be created.

Some of you will look askance at the fact that each pseudo-constructor returns an object by value. This is, you'll doubtless point out, a potential efficiency bottleneck, because each by-value return implies the cost of a constructor to create the returned object and a destructor to destroy it. This cost can be eliminated by having each makeNumber function call new:

```
UPNumber * UPNumber::makeNumber()
{ return new UPNumber; }
```

```
UPNumber * UPNumber::makeNumber(int initValue)
{ return new UPNumber(initValue); }

UPNumber * UPNumber::makeNumber( double
                                 initValue)
{ return new UPNumber(initValue); }

UPNumber * UPNumber::makeNumber( const
                                 UPNumber& rhs)
{ return new UPNumber(rhs); }
```

Of course, callers of `makeNumber` must now remember to call `delete` on the pointers they thus receive. The gain in efficiency must therefore be balanced against the risk of a memory leak.

In my next column, we'll continue our examination of how we can limit the number of instantiations of a class.

References:

E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns*, Addison-Wesley, 1995.