8 Object Factories

Object-oriented programs use inheritance and virtual functions to achieve powerful abstractions and good modularity. By postponing until runtime the decision regarding what exact function will be called, polymorphism promotes binary code reuse and extensibility. The runtime system dispatches virtual member functions to the appropriate derived object automatically, allowing you to implement complex behavior in terms of polymorphic primitives.

You can find this kind of paragraph in any book teaching object-oriented techniques. The reason it is repeated here is to contrast the nice state of affairs in "steady mode" with the unpleasant "initialization mode" situation in which you must *create* objects in a polymorphic way.

In the steady state, you already hold pointers or references to polymorphic objects, and you can invoke member functions against them. Their dynamic type is well known (although the caller might not know it). However, there are cases when you need to have the same flexibility in creating objects—subject to the paradox of "virtual constructors." You need virtual constructors when the information about the object to be created is inherently dynamic and cannot be used directly with C++ constructs.

Most often, polymorphic objects are created on the free store by using the `new` operator:

```
class Base { ... };
class Derived : public Base { ... };
class AnotherDerived : public Base { ... };
...
// Create a Derived object and assign it to a pointer to Base
Base* pB = new Derived;
```

The issue here is the actual `Derived` type name appearing in the invocation of the `new` operator. In a way, `Derived` here is much like the magic numeric constants we are advised not to use. If you want to create an object of the type `AnotherDerived`, you have to go to the actual statement and replace `Derived` with `AnotherDerived`. You cannot make the `new` operator act more dynamically: You must pass it a type, and that type must be exactly known at compile time.

This marks a fundamental difference between creating objects and invoking virtual member functions in C++. Virtual member functions are fluid, dynamic—you can change their behavior without changing the call site. In contrast, each object creation is a stumbling block of statically bound, rigid code. One of the effects is that invoking virtual functions binds the caller to the interface only (the base class). Object orientation tries to break dependency on the actual concrete type. However, at least in C++, object creation binds the caller to the most derived, concrete class.

Actually, it makes a lot of conceptual sense that things are this way: Even in everyday life, creating something is very different from dealing with it. You are supposed, then, to know exactly what you want to do when you embark on creating an object. However, sometimes

- You want to leave this exact knowledge up to another entity. For instance, instead of invoking `new` directly, you might call a virtual function `Create` of some higher-level object, thus allowing clients to change behavior through polymorphism.
- You do have the type knowledge, but not in a form that's expressible in C++. For instance, you might have a string containing `"Derived"`, so you actually know you have to create an object of type `Derived`, but you cannot pass a string containing a type name to `new` instead of a type name.

These two issues are the fundamental problems addressed by object factories, which we'll discuss in detail in this chapter. The topics of this chapter include the following:

- Examples of situations in which object factories are needed
- Why virtual constructors are inherently hard to implement in C++
- How to create objects by substituting values for types
- An implementation of a generic object factory

By the end of this chapter, we'll put together a generic object factory. You can customize the generic factory to a large degree—by the type of product, the creation method, and the product identification method. You can combine the factory thus created with other components described in this book, such as `Singleton` (Chapter 6)—for creating an application-wide object factory—and `Functor` (Chapter 5)—for tweaking factory behavior. We'll also introduce a clone factory, which is able to duplicate objects of any type.

8.1 The Need for Object Factories

There are two basic cases in which object factories are needed. The first occurs when a library needs not only to manipulate user-defined objects, but also to create them. For example, imagine you develop a framework for multiwindow document editors. Because you want the framework to be easily extensible, you provide an abstract class `Document` from which users can derive classes such as `TextDocument` and `HTMLDocument`. Another framework component may be a `DocumentManager` class that keeps the list of all open documents.

A good rule to introduce is that each document that exists in the application should be known by the `DocumentManager`. Therefore, creating a new document is tightly coupled with adding it to DocumentManager's list of documents. When two operations are so coupled, it is best to put them in the same function and never perform them in separation:

```
class DocumentManager
{
   ...
public:
   Document* NewDocument();
private:
   virtual Document* CreateDocument() = 0;
   std::list<Document*> listOfDocs_;
};

Document* DocumentManager::NewDocument()
{
   Document* pDoc = CreateDocument();
   listOfDocs_.push_back(pDoc);
   ...
   return pDoc;
}
```

The CreateDocument member function replaces a call to new. NewDocument cannot use the new operator because the concrete document to be created is not known by the time DocumentManager is written. In order to use the framework, programmers will derive from DocumentManager and override the virtual member function CreateDocument (which is likely to be pure). The GoF book (Gamma et al. 1995) calls CreateDocument a *factory method.*

Because the derived class knows exactly the type of document to be created, it can invoke the new operator directly. This way, you can remove the type knowledge from the framework and have the framework operate on the base class Document only. The override is very simple and consists essentially of a call to new; for example:

```
Document* GraphicDocumentManager::CreateDocument()
{
   return new GraphicDocument;
}
```

Alternatively, an application built with this framework might support creation of multiple document types (for instance, bitmapped graphics and vectorized graphics). In that case, the overridden CreateDocument function might display a dialog to the user asking for the specific type of document to be created.

Thinking of opening a document previously saved on disk in the framework just outlined brings us to the second—and more complicated—case in which an object factory may be needed. When you save an object to a file, you must save its actual type in the form of a string, an integral value, an identifier of some sort. However, although the type information exists, its *form* does not allow you to create C++ objects.

The general concept underlying this situation is creating objects whose type information is genuinely postponed to runtime: entered by the end user, read from a persistent storage or network connection, or the like. Here the binding of types to values is pushed even further than in the case of polymorphism: When using polymorphism, the entity manipulating an object does not know its type exactly; however, the object itself is of a well-determined type. When reading objects from some storage, the type comes "alone" at runtime. You must transform type information into an object. Finally, you must read the object from the storage, which is easy once an empty object is created, by invoking a virtual function.

Creating objects from "pure" type information, and consequently adapting *dynamic* information to *static* C++ types, is an important issue in building object factories. Let's focus on it in the next section.

8.2 Object Factories in C++: Classes and Objects

In order to come up with a solution, we need a good grasp of the problem. This section tries to answer the following questions: Why are C++ constructors so rigid? Why don't we have flexible means to create objects in the language itself?

Interestingly, seeking an answer to this question takes us directly to fundamental decisions about C++'s type system. In order to find out why a statement like

```
Base* pB = new Derived;
```

is so rigid, we must answer two related questions: What is a class, and what is an object? This is because the culprit in the given statement is `Derived`, which is a class name, and we'd like it to be a value, that is, an object.

In C++, classes and objects are different beasts. Classes are what the programmer creates, and objects are what the program creates. You cannot create a new class at runtime, and you cannot create an object at compile time. Classes don't have first-class status: You cannot copy a class, store it in a variable, or return it from a function.

In contrast, there are languages where classes *are* objects. In those languages, some objects with certain properties are simply considered classes by convention. Consequently, in those languages, you *can* create new classes at runtime, copy a class, store it in a variable, and so on. If C++ were such a language, you could have written code like the following:

```
// Warning—this is NOT C++
// Assumes Class is a class that's also an object
Class Read(const char* fileName);
Document* DocumentManager::OpenDocument(const char* fileName)
{
   Class theClass = Read(fileName);
   Document* pDoc = new theClass;
   ...
}
```

That is, we could pass a variable of type `Class` to the `new` operator. In such a paradigm, passing a known class name to `new` is the same as using a hardcoded constant.

Such dynamic languages trade off some type safety and performance for the sake of flexibility, because static typing is an important source of optimization. C++ took the opposite approach, sticking to a static type system, yet trying to provide as much flexibility as possible in this framework.

The bottom line is, creating object factories is a complicated problem in C++. In C++ there is a fracture between types and values: A value has a type attribute, but a type cannot exist on its own. If you want to create an object in a totally dynamic way, you need a means to express and pass around a "pure" type and build a value from it on demand. Because you cannot do this, you somehow must represent types as objects—integers, strings, and so on. Then, you must employ some trick to exchange the value for the right type, and finally to use that type to create an object. This object-type-object trade is fundamental for object factories in statically typed languages.

We will call the object that identifies a type a *type identifier.* (Don't confuse it with `std::typeid`.) The type identifier helps the factory in creating the appropriate type of object. As will be shown, sometimes you make the type identifier–object exchange without knowing exactly what you have or what you will get. It's like in fairy tales: You don't know exactly how the token works (and it's sometimes dangerous to try to figure it out), but you pass it to the wizard, who gives you a valuable object in exchange. The details of how the magic happens must be encapsulated in the wizard . . . the factory, that is.

We will explore a simple factory that solves a concrete problem, try various implementations of it, and then extract the generic part of it into a class template.

8.3 Implementing an Object Factory

Say you write a simple drawing application, allowing editing of simple vectorized drawings consisting of lines, circles, polygons, and so on.[1] In a classic object-oriented manner, you define an abstract `Shape` class from which all your figures will derive:

```
class Shape
{
public:
   virtual void Draw() const = 0;
   virtual void Rotate(double angle) = 0;
   virtual void Zoom(double zoomFactor) = 0;
   ...
};
```

You might then define a class `Drawing` that contains a complex drawing. A `Drawing` essentially holds a collection of pointers to `Shape`—such as a list, a vector, or a hierarchical structure—and provides operations to manipulate the drawing as a whole. Two typical operations you might want to do are saving a drawing as a file and loading a drawing from a previously saved file.

Saving shapes is easy: Just provide a pure virtual function like `Shape::Save(std::ostream&)`. Then the `Drawing::Save` operation might look like this:

```
class Drawing
{
public:
```

[1]This "Hello, world" of design is a good basis for C++ interview questions. Although many candidates manage to conceive such a design, few of them know how to implement loading files, which is a rather important operation.

```
   void Save(std::ofstream& outFile);
   void Load(std::ifstream& inFile);
   ...
};

void Drawing::Save(std::ofstream& outFile)
{
   write drawing header
   for (each element in the drawing)
   {
      (current element)->Save(outFile);
   }
}
```

The Shape-Drawing example just described is often encountered in C++ books, including Bjarne Stroustrup's classic (Stroustrup 1997). However, most C++ introductory books stop when it comes to loading graphics from a file, exactly because the nice model of having separate drawing objects breaks. Explaining the gory details of reading objects makes for a big parenthesis, which is often understandably avoided. On the other hand, this is exactly what we want to implement, so we have to bite the bullet. A straightforward implementation is to require each Shape-derived object to save an integral identifier at the very beginning. Each object should have its own unique ID. Then reading the file would look like this:

```
// a unique ID for each drawing object type
namespace DrawingType
{
const int
   LINE = 1,
   POLYGON = 2,
   CIRCLE = 3
};

void Drawing::Load(std::ifstream& inFile)
{
   // error handling omitted for simplicity
   while (inFile)
   {
      // read object type
      int drawingType;
      inFile >> drawingType;

      // create a new empty object
      Shape* pCurrentObject;
      switch (drawingType)
      {
         using namespace DrawingType;
      case LINE:
         pCurrentObject = new Line;
         break;
      case POLYGON:
         pCurrentObject = new Polygon;
         break;
```

```
            case CIRCLE:
               pCurrentObject = new Circle;
               break;
            default:
               handle error—unknown object type
            }
            // read the object's contents by invoking a virtual fn
            pCurrentObject->Read(inFile);
            add the object to the container
         }
      }
```

This is indeed an object factory. It reads a type identifier from the file, creates an object of the appropriate type based on that identifier, and invokes a virtual function that loads that object from the file. The only problem is that it breaks the most important rules of object orientation:

- It performs a `switch` based on a type tag, with the associated drawbacks, which is exactly what object-oriented programs try to eliminate.
- It collects in a single source file knowledge about all `Shape`-derived classes in the program, which again you must strive to avoid. For one thing, the implementation file of `Drawing::Save` must include all headers of all possible shapes, which makes it a bottleneck of compile dependencies and maintenance.
- It is hard to extend. Imagine adding a new shape, such as `Ellipse`, to the system. In addition to creating the class itself, you must add a distinct integral constant to the namespace `DrawingType`, you must write that constant when saving an `Ellipse` object, and you must add a label to the `switch` statement in `Drawing::Save`. This is an awful lot more than what the architecture promised—total insulation between classes—and all for the sake of a single function!

We'd like to create an object factory that does the job without having these disadvantages. One practical goal worth pursuing is to break the `switch` statement apart, so we can put the `Line` creation statement in file implementation for `Line`, and do the same for `Polygon` and `Circle`.

A common way to keep together and manipulate pieces of code is to work with pointers to functions, as discussed at length in Chapter 5. The unit of customizable code here (each of the entries in the `switch` statement) can be abstracted in a function with the signature

```
      Shape* CreateConcreteShape();
```

The factory keeps a collection of pointers to functions with this signature. In addition, there has to be a correspondence between the IDs and the pointer to the function that creates the appropriate object. Thus, what we need is an *associative collection*—a map. A map offers access to the appropriate function given the type identifier, which is precisely what the `switch` statement offers. In addition, the map offers the scalability that the `switch` statement, with its compile-time fixed structure, cannot provide. The map can grow at runtime—you can add entries (tuples of IDs and pointers to functions) dynamically, which is exactly what we need. We can start with an empty map and have each Shape-derived object add an entry to it.

Why not use a vector? IDs are integral numbers, so we can keep a vector and have the ID be the index in the vector. This would be simpler and faster, but a map is better here. The map doesn't require its indices to be adjacent, plus it's more general—vectors work only with integral indices, whereas maps accept any ordered type as an index. This point will become important when we generalize our example.

We can start designing a ShapeFactory class, which has the responsibility of managing the creation of all Shape-derived objects. In implementing ShapeFactory, we will use the map implementation found in the standard library, std::map:

```
class ShapeFactory
{
public:
   typedef Shape* (*CreateShapeCallback)();
private:
   typedef std::map<int, CreateShapeCallback> CallbackMap;
public:
   // Returns 'true' if registration was successful
   bool RegisterShape(int ShapeId,
      CreateShapeCallback CreateFn);
   // Returns 'true' if the ShapeId was registered before
   bool UnregisterShape(int ShapeId);
   Shape* CreateShape(int ShapeId);
private:
   CallbackMap callbacks_;
};
```

This is a basic design of a scalable factory. The factory is scalable because you don't have to modify its code each time you add a new Shape-derived class to the system. ShapeFactory divides responsibility: Each new shape has to register itself with the factory by calling RegisterShape and passing it its integral identifier and a pointer to a function that creates an object. Typically, the function has a single line and looks like this:

```
Shape* CreateLine()
{
   return new Line;
}
```

The implementation of Line also must register this function with the ShapeFactory that the application uses, which is typically a globally accessible object.[2] The registration is usually performed with startup code. The whole connection of Line with the Shape Factory is as follows:

```
// Implementation module for class Line
// Create an anonymous namespace
//  to make the function invisible from other modules
namespace
```

[2]This brings us to the link between object factories and singletons. Indeed, more often than not, factories *are* singletons. Later in this chapter is a discussion of how to use factories with the singletons implemented in Chapter 6.

```
{
   Shape* CreateLine()
   {
      return new Line;
   }
   // The ID of class Line
   const int LINE = 1;
   // Assume TheShapeFactory is a singleton factory
   // (see Chapter 6)
   const bool registered =
      TheShapeFactory::Instance().RegisterShape(
         LINE, CreateLine);
}
```

Implementing the `ShapeFactory` is easy, given the amenities `std::map` has to offer. Basically, `ShapeFactory` member functions forward only to the `callbacks_` member:

```
bool ShapeFactory::RegisterShape(int shapeId,
   CreateShapeCallback createFn)
{
   return callbacks_.insert(
      CallbackMap::value_type(shapeId, createFn)).second;
}

bool ShapeFactory::UnregisterShape(int shapeId)
{
   return callbacks_.erase(shapeId) == 1;
}
```

If you're not very familiar with the `std::map` class template, the previous code might need a bit of explanation:

- `std::map` holds pairs of keys and data. In our case, keys are integral shape IDs, and the data consists of a pointer to function. The type of our pair is `std::pair<const int, CreateShapeCallback>`. You must pass an object of this type when you call `insert`. Because that's a lot to type, it's better to use the typedef found inside `std::map`, which provides a handy name—`value_type`—for that pair type. Alternatively, you can use `std::make_pair`.
- The `insert` member function we called returns another pair, this time containing an iterator (which refers to the element just inserted) and a `bool` that is `true` if the value didn't exist before, and `false` otherwise. The `.second` field access after the call to `insert` selects this `bool` and returns it in a single stroke, without us having to create a named temporary.
- `erase` returns the number of elements erased.

The `CreateShape` member function simply fetches the appropriate pointer to a function for the ID passed in, and calls it. In the case of an error, it throws an exception. Here it is:

```
Shape* ShapeFactory::CreateShape(int shapeId)
{
   CallbackMap::const_iterator i = callbacks_.find(shapeId);
```

```
      if (i == callbacks_.end())
      {
         // not found
         throw std::runtime_error("Unknown Shape ID");
      }
      // Invoke the creation function
      return (i->second)();
   }
```

Let's see what this simple class brought us. Instead of relying on the large, know-it-all `switch` statement, we obtained a dynamic scheme requiring each type of object to register itself with the factory. This moves the responsibility from a centralized place to each concrete class, where it belongs. Now whenever you define a new `Shape`-derived class, you can just *add* files instead of *modifying* files.

8.4 Type Identifiers

The only problem that remains is the management of type identifiers. Still, adding type identifiers requires a fair amount of discipline and centralized control. Whenever you add a new shape class, you must check all the existing type identifiers and add one that doesn't clash with them. If a clash exists, the second call to `RegisterShape` for the same ID fails, and you won't be able to create objects of that type.

We can solve this problem by choosing a more generous type than `int` for expressing the type identifier. Our design doesn't require integral types, only types that can be keys in a `map`, that is, types that support `operator==` and `operator<`. (That's why we can be happy we chose maps instead of vectors.) For example, we can store type identifiers as strings and establish the convention that each class is represented by its name: `Line`'s identifier is "Line", `Polygon`'s identifier is "Polygon", and so forth. This minimizes the chance of clashing names because class names are unique.

If you enjoy spending your weekends studying C++, maybe the previous paragraph rang a bell for you. Let's use `type_info`! The `std::type_info` class is part of the runtime type information (RTTI) provided by C++. You get a reference to a `std::type_info` by invoking the `typeid` operator on a type or an expression. What seems nice is that `std::type_info` provides a `name` member function that returns a `const char*` pointing to a human-readable name of the type. For your compiler, you might have seen that `typeid(Line).name()` points to the string "class Line", which is exactly what we wanted.

The problem is, this does not apply to all C++ compiler implementations. The way `type_info::name` is defined makes it unsuitable for anything other than debugging purposes (like printing it in a debug console). There is no guarantee that the string is the actual class name, and worse, there is no guarantee that the string is unique throughout the application. (Yes, you can have two classes that have the same name according to `std::type_info::name`.) And the shotgun argument is that there's no guarantee that the type name will be unique in *time.* There is no guarantee that `typeid(Line).name()` points to the same string when the application is run twice. Implementing persistence is an important application of factories, and `std::type_info::name` is *not* persistent. This all makes `std::type_info` deceptively close to being useful for our object factory, but it is not a real solution.

Back to the management of type identifiers. A decentralized solution for generating type identifiers is to use a unique value generator—for instance, a random number or random string generator. You would use this generator each time you add a new class to the program, then hardcode that random value in the source file and never change it.[3] This sounds like a brittle solution, but think of it this way: If you have a random string generator that has a 10^{-20} probability of repeating a value in a thousand years, you get a rate of errors smaller than that of a program using a "perfect" factory.

The only conclusion that can be drawn here is that type identifier management is not the business of the object factory itself. Because C++ cannot guarantee a unique, persistent type ID, type ID management becomes an extra-linguistic issue that must be left to the programmers.

We have described all the elements in a typical object factory, and we have a prototype implementation. It's time now for the next step—the step from concrete to abstract. Then, enriched with new insights, we'll go back to concrete.

8.5 Generalization

Let's enumerate the elements involved in our discussion of object factories. This gives us the intellectual workout necessary for putting together a generic object factory.

- *Concrete product.* A factory delivers some product in the form of an object.
- *Abstract product.* Products inherit a base type (in our example, `Shape`). A product is an object whose type belongs to a hierarchy. The base type of the hierarchy is the abstract product. The factory behaves polymorphically in the sense that it returns a pointer to the abstract product, without conveying knowledge of the concrete product type.
- *Product type identifier.* This is the object that identifies the type of the concrete product. As discussed, you have to have a type identifier to create a product because of the static C++ type system.
- *Product creator.* The function or functor is specialized for creating exactly one type of object. We modeled the product creator with a pointer to function.

The generic factory will orchestrate these elements to provide a well-defined interface, as well as defaults for the most used cases.

It seems that each of the notions just enumerated will transform into a template parameter of a `Factory` template class. There's only one exception: The concrete product doesn't have to be known to the factory. Had this been the case, we'd have different

[3]Microsoft's COM factory uses such a method. They have an algorithm for generating unique 128-bit identifiers (called globally unique identifiers, or GUIDs) for COM objects. The algorithm relies on the uniqueness of the network card serial number or, in the absence of a card, the date, time, and other highly variable machine states.

Factory types for each concrete product we're adding, and we are trying to keep Factory insulated from the concrete types. We want only different Factory types for different abstract products.

This being said, let's write down what we've grabbed so far:

```
template
<
   class AbstractProduct,
   typename IdentifierType,
   typename ProductCreator
>
class Factory
{
public:
   bool Register(const IdentifierType& id, ProductCreator creator)
   {
      return associations_.insert(
         AssocMap::value_type(id, creator)).second;
   }
   bool Unregister(const IdentifierType& id)
   {
      return associations_.erase(id) == 1;
   }
   AbstractProduct* CreateObject(const IdentifierType& id)
   {
      typename AssocMap::const_iterator i =
         associations_.find(id);
      if (i != associations_.end())
      {
         return (i->second)();
      }
      handle error
   }
private:
   typedef std::map<IdentifierType, AbstractProduct>
      AssocMap;
   AssocMap associations_;
};
```

The only thing left out is error handling. If we didn't find a creator registered with the factory, should we throw an exception? Return a null pointer? Terminate the program? Dynamically load some library, register it on the fly, and retry the operation? The actual decision depends very much on the concrete situation; any of these actions makes sense in some cases.

Our generic factory should let the user customize it to do any of these actions and should provide a reasonable default behavior. Therefore, the error handling code should be pulled out of the CreateObject member function into a separate FactoryError policy (see Chapter 1). This policy defines only one function, OnUnknownType, and Factory gives that function a fair chance (and enough information) to make any sensible decision.

The policy defined by FactoryError is very simple. FactoryError prescribes a template of two parameters: `IdentifierType` and `AbstractProduct`. If `FactoryErrorImpl` is an implementation of FactoryError, then the following expression must apply:

```
FactoryErrorImpl<IdentifierType, AbstractProduct> factoryErrorImpl;
IdentifierType id;
AbstractProduct* pProduct = factoryErrorImpl.OnUnknownType(id);
```

`Factory` uses `FactoryErrorImpl` as a last-resort solution: If `CreateObject` cannot find the association in its internal map, it uses `FactoryErrorImpl<IdentifierType,AbstractProduct>::OnUnknownType` for fetching a pointer to the abstract product. If `OnUnknownType` throws an exception, the exception propagates out of `Factory`. Otherwise, `CreateObject` simply returns whatever `OnUnknownType` returned.

Let's code these additions and changes (shown in bold):

```
template
<
   class AbstractProduct,
   typename IdentifierType,
   typename ProductCreator,
   template<typename, class>
      class FactoryErrorPolicy
>
class Factory
   : public FactoryErrorPolicy<IdentifierType, AbstractProduct>
{
public:
   AbstractProduct* CreateObject(const IdentifierType& id)
   {
      typename AssocMap::const_iterator i = associations_.find(id);
      if (i != associations_.end())
      {
         return (i->second)();
      }
      return OnUnknownType(id);
   }
private:
   ... rest of functions and data as above ...
};
```

The default implementation of FactoryError throws an exception. This exception's class is best made distinct from all other types so that client code can detect it separately and make appropriate decisions. Also, the class should inherit one of the standard exception classes so that the client can catch all kinds of errors with one `catch` block. FactoryError defines a nested exception class (called `Exception`)[4] that inherits `std::runtime_error`.

[4]There is no need to make the name distinctive (like `FactoryException`), because the type is already inside class template `Factory`.

```
template <class IdentifierType, class ProductType>
class DefaultFactoryError
{
public:
   class Exception : public std::exception
      {
      public:
         Exception(const IdentifierType& unknownId)
            : unknownId_(unknownId)
         {
         }
         virtual const char* what()
         {
            return "Unknown object type passed to Factory.";
         }
         const IdentifierType GetId()
         {
            return unknownId_;
         };
      private:
         IdentifierType unknownId_;
      };
protected:
   StaticProductType* OnUnknownType(const IdentifierType& id)
   {
      throw Exception(id);
   }
};
```

Other, more advanced implementations of FactoryError can look up the type identifier and return a pointer to a valid object, return a null pointer (if use of exceptions is undesirable), throw some exception object, or terminate the program. You can tweak the behavior by defining new FactoryError implementations and specifying them as the fourth argument of `Factory`.

8.6 Minutiae

Actually, Loki's `Factory` implementation does not use `std::map`. It uses a drop-in replacement for map, `AssocVector`, which is optimized for rare inserts and frequent lookups, the typical usage pattern of `Factory`. `AssocVector` is described in detail in Chapter 11.

In an initial draft of `Factory`, the map type was customizable by being a template parameter. However, just too often `AssocVector` fits the bill exactly; in addition, using standard containers as template template parameters is not, well, standard. This is because standard container implementers are free to add more template arguments, as long as they provide defaults for them.

Let's focus now on the `ProductCreator` template parameter. Its main requirement is that it should have functional behavior (accept `operator()` taking no arguments) and return a pointer convertible to `AbstractProduct*`. In the concrete implementation shown earlier, ProductCreator was a simple pointer to a function. This suffices if all we need is to create objects by invoking new, which is the most common case. Therefore, we choose

```
AbstractProduct* (*)()
```

as the default type for ProductCreator. The type looks a bit like a confusing emoticon because its name is missing. If you put a name after the asterisk within the parentheses, like so,

```
AbstractProduct* (*PointerToFunction)()
```

the type reveals itself as a pointer to a function taking no parameters and returning a pointer to AbstractProduct. If this still looks unfamiliar to you, you may want to refer to Chapter 5, which includes a discussion on pointers to functions.

By the way, speaking of that chapter, there is a very interesting template parameter you can pass to Factory as ProductCreator, namely Functor<AbstractProduct*>. If you choose this, you gain great flexibility: You can create objects by invoking a simple function, a member function, or a functor, and bind appropriate parameters to any of them. The glue code is provided by Functor.

Our Factory class template declaration now looks like this:

```
template
<
   class AbstractProduct,
   class IdentifierType,
   class ProductCreator = AbstractProduct* (*)(),
   template<typename, class>
      class FactoryErrorPolicy = DefaultFactoryError
>
class Factory;
```

Our Factory class template is now ready to be of use.

8.7 Clone Factories

Although genetic factories producing clones of the universal soldier are quite a scary prospect, cloning C++ objects is a harmless and useful activity most of the time. Here the goal is slightly different from what we dealt with so far: We don't have to create objects from scratch anymore. We have a pointer to a polymorphic object, and we'd like to create an exact copy of it. Because we don't exactly know the type of the polymorphic object, we don't exactly know what new object to create, and this is the actual issue.

Because we do have an object at hand, we can apply classic polymorphism. Thus, the usual idiom used for object cloning is to declare a virtual Clone member function in the base class and to have each derived class override it. Here's an example using our geometric shapes hierarchy:

```
class Shape
{
```

```
public:
   virtual Shape* Clone() const = 0;
   ...
};

class Line : public Shape
{
public:
   virtual Line* Clone() const
   {
      return new Line(*this);
   }
   ...
};
```

The reason that `Line::Clone` does not return a pointer to `Shape` is that we took advantage of a C++ feature called *covariant return types.* Because of covariant return types, you can return a pointer to a derived class instead of a pointer to the base class from an overridden virtual function. From now on, the idiom goes, you must implement a similar `Clone` function for each class you add to the hierarchy. The contents of the functions are the same: You create a `Polygon`, you return a `new Polygon(*this)`, and so on.

This idiom works, but it has a couple of major drawbacks:

- If the base class wasn't designed to be cloneable (didn't declare a virtual function equivalent to `Clone`) and is not modifiable, the idiom cannot be applied. This is the case when you write an application using a class library that requires you to derive from its base classes.
- Even if all the classes are changeable, the idiom requires a high amount of discipline. Forgetting to implement `Clone` in some derived classes will remain undetected by the compiler and may cause runtime behavior ranging from bizarre to pernicious.

The first point is obvious; let's discuss the second one. Imagine you derived a class `DottedLine` from `Line` and forgot to override `DottedLine::Clone`. Now say you have a pointer to a `Shape` that actually points to a `DottedLine`, and you invoke `Clone` on it:

```
Shape* pShape;
...
Shape* pDuplicateShape = pShape->Clone();
```

The `Line::Clone` function will be invoked, returning a `Line`. This is a very unfortunate situation because you assume `pDuplicateShape` to have the same dynamic type as `pShape`, when in fact it doesn't. This might lead to a lot of problems, from drawing unexpected types of lines to crashing the application.

There's no solid way to mitigate this second problem. You can't say in C++: "I define this function, and I require any direct or indirect class inheriting it to override it." You must shoulder the painful repetitive task of overriding `Clone` in every shape class, and you're doomed if you don't.

If you agree to complicate the idiom a bit, you can get an acceptable runtime check.

Make Clone a public nonvirtual function. From inside it call a *private virtual* function called, say, DoClone, and then enforce the equality of the dynamic types. The code is simpler than the explanation:

```
class Shape
{
   ...
public:
   Shape* Clone() const  //nonvirtual
   {
      // delegate to DoClone
      Shape* pClone = DoClone();
      // Check for type equivalence
      // (could be a more sophisticated test than assert)
      assert(typeid(*pClone) == typeid(*this));
      return pClone;
   }
private:
   virtual Shape* DoClone() const = 0; // private
};
```

The only downside is that you cannot use covariant return types anymore.

Shape derivees would always override DoClone, leave it private so clients cannot call it, and leave Clone alone. Clients use Clone only, which performs the runtime check. As you certainly figured out, programming errors, such as overriding Clone or making DoClone public, can still sneak in.

Don't forget that, no matter what, if you cannot change all the classes in the hierarchy (the hierarchy is *closed*) and if it wasn't designed to be cloneable, you don't have any chance of implementing this idiom. This is quite a dismissive argument in many cases, so we should look for alternatives.

Here a special object factory may be of help. It leads to a solution that doesn't have the two problems mentioned earlier, at the cost of a slight performance hit—instead of a virtual call, there is a map lookup plus a call via a pointer to a function. Because the number of classes in an application is never really big (they are written by people, aren't they?), the map tends to be small, and the hit should not be significant.

It all starts from the idea that in a clone factory, the type identifier and the product have the same type. You receive the object to be duplicated as a type identifier and pass as output a new object that is a copy of the type identifier. To be more precise, they're not quite the same type: A cloning factory's IdentifierType is a *pointer to* AbstractProduct. The exact deal is that you pass a pointer to the clone factory, and you get back another pointer, which points to a cloned object.

But what's the key in the map? It can't be a pointer to AbstractProduct because you don't need as many entries as the objects we have. You need only one entry per *type* of object to be cloned, which brings us again to the std::type_info class. The fact that the type identifier passed when the factory is asked to create a new object is different from the type identifier that's stored in the association map makes it impossible for us to reuse the code we wrote so far. Another consequence is that the product creator now needs the pointer to the object to be cloned; in the factory from scratch above, no parameter was needed.

Let's recap. The clone factory gets a pointer to an AbstractProduct. It applies the typeid operator to the pointed-to object and obtains a reference to a std::type_info object. It then looks up that object in its private map. (The before member function of std::type_info introduces an ordering over the set of std::type_info objects, which makes it possible to use a map and perform fast searches.) If an entry is not found, an exception is thrown. If it is, the product creator will be invoked, with the pointer to the AbstractProduct passed in by the user.

Because we already have the Factory class template handy, implementing the Clone-Factory class template is a simple exercise. (You can find it in Loki.) There are a few differences and new elements:

- CloneFactory uses TypeInfo instead of std::type_info. The class TypeInfo, discussed in Chapter 2, is a wrapper around a pointer to std::type_info, having the purpose of defining proper initialization, assignment, operator==, and operator<, which are all needed by the map. The first operator delegates to std::type_info::operator==; the second operator delegates to std::type_info::before.
- There is no IdentifierType anymore because the identifier type is implicit.
- The ProductCreator template parameter defaults to AbstractProduct*(*) (Abstract-Product*).
- The IdToProductMap is now AssocVector<TypeInfo,ProductCreator>.

The synopsis of CloneFactory is as follows:

```
template
<
   class AbstractProduct,
   class ProductCreator =
      AbstractProduct* (*)(AbstractProduct*),
   template<typename, class>
      class FactoryErrorPolicy = DefaultFactoryError
>
class CloneFactory
{
public:
   AbstractProduct* CreateObject(const AbstractProduct* model);
   bool Register(const TypeInfo&,
      ProductCreator creator);
   bool Unregister(const TypeInfo&);
private:
   typedef AssocVector<TypeInfo, ProductCreator>
      IdToProductMap;
   IdToProductMap associations_;
};
```

The CloneFactory class template is a complete solution for cloning objects belonging to closed class hierarchies (that is, class hierarchies that you cannot modify). Its simplicity and effectiveness stem from the conceptual clarifications made in the previous sections and from the runtime type information that C++ provides through `typeid` and `std::type_info`. Had RTTI not existed, clone factories would have been much more awkward to implement—in fact, so awkward that putting them together wouldn't have made much sense in the first place.

8.8 Using Object Factories with Other Generic Components

Chapter 6 introduced the `SingletonHolder` class, which was designed to provide specific services to your classes. Because of the global nature of factories, it is natural to use `Factory` with `SingletonHolder`. They are very easy to combine by using `typedef`. For instance:

```
typedef SingletonHolder< Factory<Shape, std::string> > ShapeFactory;
```

Of course, you can add arguments to any `SingletonHolder` or `Factory` to choose different trade-offs and design decisions, but it's all in one place. From now on, you can isolate a bunch of important design choices in one place and use `ShapeFactory` throughout the code. Within the simple type definition just shown, you can select the way the factory works and the way the singleton works, thus exploiting all the combinations between the two. With a single line of declarative code, you direct the compiler to generate the right code for you and nothing more, just like at runtime you'd call a function with various parameters to perform some action in different ways. Because in our case it all happens at compile time, the emphasis is more on design decisions than on runtime behavior. Of course, runtime behavior is affected as well, but in a more global and subtle way. By writing "regular" code, you specify what's going to happen at runtime. When you write a type definition like the one above, you specify what's going to happen during compile time—in fact, you kind of call code-generation functions at compile time, passing arguments to them.

As alluded to in the beginning of this chapter, an interesting combination is to use `Factory` with `Functor`:

```
typedef SingletonHolder
<
   Factory
   <
      Shape, std::string, Functor<Shape*>
   >
>
ShapeFactory;
```

This gives you great flexibility in creating objects, by leveraging the power of `Functor` (for which implementation we took great pains in Chapter 5). You can now create `Shapes` in almost any way imaginable by registering various `Functors` with the `Factory`, and the whole thing is a `Singleton`.

8.9 Summary

Object factories are an important component of programs using polymorphism. They help in creating objects when their type is either not available or available in a form that's incompatible for use with language constructs.

Object factories are used mostly in object-oriented frameworks and libraries, as well as in various object persistence and streaming schemes. The latter case was analyzed in depth with a concrete example. The solution discussed essentially distributes a switch on type across multiple implementation files, thus achieving low coupling. Although the factory remains a central authority that creates objects, it doesn't have to collect knowledge about all the static types in a hierarchy. Instead, it's the responsibility of each type to register itself with the factory. This marks a fundamental difference between the "wrong" and the "right" approach.

Type information cannot be easily transported at runtime in C++. This is a fundamental feature of the family of languages to which C++ belongs. Because of this, type identifiers that represent types have to be used instead. They are associated with creator objects that are callable entities, as described in Chapter 5 on generalized functors (pointers to functions or functors). A concrete object factory starting from these ideas was implemented and was then generalized into a class template.

Finally, we discussed clone factories (factories that are able to duplicate polymorphic objects).

8.10 `Factory` Class Template Quick Facts

- `Factory` declaration:

```
template
<
   class AbstractProduct,
   class IdentifierType,
   class ProductCreator = AbstractProduct* (*)(),
   template<typename, class>
      class FactoryErrorPolicy = DefaultFactoryError
>
class Factory;
```

- `AbstractProduct` is the base class of the hierarchy for which you provide the object factory.
- `IdentifierType` is the type of the "cookie" that represents a type in the hierarchy. It has to be an ordered type (able to be stored in a `std::map`). Commonly used identifier types are strings and integral types.
- `ProductCreator` is the callable entity that creates objects. This type must support `operator()` taking no parameters and returning a pointer to `AbstractProduct`. A `ProductCreator` object is always registered together with a type identifier.
- `Factory` implements the following primitives:

```
bool Register(const IdentifierType& id, ProductCreator creator);
```

Registers a creator with a type identifier. Returns `true` if the registration was successful; `false` otherwise (if there already was a creator registered with the same type identifier).

```
bool Unregister(const IdentifierType& id);
```

Unregisters the creator for the given type identifier. If the type identifier was previously registered, the function returns `true`.

```
AbstractProduct* CreateObject(const IdentifierType& id);
```

Looks up the type identifier in the internal map. If found, it invokes the corresponding creator for the type identifier and returns its result. If the type identifier is not found, the result of `FactoryErrorPolicy<IdentifierType,AbstractProduct>:: OnUnknownType` is returned. The default implementation of `FactoryErrorPolicy` throws an exception of its nested type `Exception`.

8.11 CloneFactory Class Template Quick Facts

- `CloneFactory` declaration:

```
template
<
   class AbstractProduct,
   class ProductCreator =
      AbstractProduct* (*)(ConstAbstractProduct*),
   template<typename, class>
      class FactoryErrorPolicy = DefaultFactoryError
>
class CloneFactory;
```

- `AbstractProduct` is the base class of the hierarchy for which you want to provide the clone factory.
- `ProductCreator` has the role of duplicating the object received as a parameter and returning a pointer to the clone.
- `CloneFactory` implements the following primitives:

```
bool Register(const TypeInfo&, ProductCreator creator);
```

Registers a creator with an object of type `TypeInfo` (which accepts an implicit conversion constructor from `std::type_info`). Returns `true` if the registration was successful; `false` otherwise.

```
bool Unregister(const TypeInfo& typeInfo);
```

Unregisters the creator for the given type. If the type was previously registered, the function returns `true`.

```
AbstractProduct* CreateObject(const AbstractProduct* model);
```

Looks up the dynamic type of `model` in the internal map. If found, it invokes the corresponding creator for the type identifier and returns its result. If the type identifier is not found, the result of `FactoryErrorPolicy<OrderedTypeInfo, AbstractProduct>::OnUnknownType` is returned.