---

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 which specific function will be called, polymorphism promotes binary code reuse and extensibility. The runtime system automatically dispatches virtual member functions to the appropriate derived object, 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:

Click here to view code image

```
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 the creation of 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 can duplicate objects of any type.