
8.2. Object Factories in C++: Classes and Objects

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. To find out why a statement such as

```
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 in which 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:

Click here to view code image

```
// 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 that 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 `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 a fairy tale: 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.