Item 41: Understand implicit interfaces and compile-time polymorphism

The world of object-oriented programming revolves around *explicit* interfaces and *runtime* polymorphism. For example, given this (meaningless) class,

```
class Widget {

public:

  Widget();

  virtual ~Widget();

  virtual std::size_t size() const;

  virtual void normalize();

  void swap(Widget& other);                  // see Item 25


  ...

};
```

and this (equally meaningless) function,

```
void doProcessing(Widget& w)

{

  if (w.size() > 10 && w != someNastyWidget) {

       Widget temp(w);

       temp.normalize();

       temp.swap(w);

  }

}
```

we can say this about `w` in `doProcessing`:

- Because `w` is declared to be of type `Widget`, `w` must support the `Widget` interface. We can look up this interface in the source code (e.g., the `.h` file for `Widget`) to see exactly what it looks like, so I call this an *explicit interface* — one explicitly visible in the source code.
- Because some of `Widget`'s member functions are virtual, `w`'s calls to those functions will exhibit *runtime polymorphism*: the specific function to call will be determined at runtime based on `w`'s dynamic type (see Item 37).

The world of templates and generic programming is fundamentally different. In that world, explicit interfaces and runtime polymorphism continue to exist, but they're less important. Instead, *implicit interfaces* and *compile-time polymorphism* move to the fore. To see how this is the case, look what happens when we turn `doProcessing` from a function into a function template:

```
template<typename T>
void doProcessing(T& w)
{
  if (w.size() > 10 && w != someNastyWidget) {
     T temp(w);
     temp.normalize();
     temp.swap(w);
  }
}
```

Now what can we say about `w` in `doProcessing`?

- The interface that `w` must support is determined by the operations performed on `w` in the template. In this example, it appears that `w`'s type (`T`) must support the `size`, `normalize`, and `swap` member functions; copy construction (to create `temp`); and comparison for inequality (for comparison with `someNastyWidget`). We'll soon see that this isn't quite accurate, but it's true enough for now. What's important is that the set of expressions that must be valid in order for the template to compile is the *implicit interface* that `T` must support.
- The calls to functions involving `w` such as `operator>` and `operator!=` may involve instantiating templates to make these calls succeed. Such instantiation occurs during compilation. Because instantiating function templates with different template parameters leads to different functions being called, this is known as *compile-time polymorphism*.

Even if you've never used templates, you should be familiar with the difference between runtime and compile-time polymorphism, because it's similar to the difference between the process of determining which of a set of overloaded functions should be called (which takes place during compilation) and dynamic binding of virtual function calls (which takes place at runtime). The difference between explicit and implicit interfaces is new to templates, however, and it bears closer examination.

An explicit interface typically consists of function signatures, i.e., function names, parameter types, return types, etc. The `Widget` class public interface, for example,

```
class Widget {
public:
  Widget();
  virtual ~Widget();

  virtual std::size_t size() const;
```

```
  virtual void normalize();

  void swap(Widget& other);

};
```

consists of a constructor, a destructor, and the functions `size`, `normalize`, and `swap`, along with the parameter types, return types, and constnesses of these functions. (It also includes the compiler-generated copy constructor and copy assignment operator — see Item 5.) It could also include typedefs and, if you were so bold as to violate Item 22's advice to make data members private, data members, though in this case, it does not.

An implicit interface is quite different. It is not based on function signatures. Rather, it consists of valid *expressions*. Look again at the conditional at the beginning of the `doProcessing` template:

```
template<typename T>

void doProcessing(T& w)

{

  if (w.size() > 10 && w != someNastyWidget) {

  ...
```

The implicit interface for `T` (`w`'s type) appears to have these constraints:

- It must offer a member function named `size` that returns an integral value.
- It must support an `operator!=` function that compares two objects of type `T`. (Here, we assume that `someNastyWidget` is of type `T`.)

Thanks to the possibility of operator overloading, neither of these constraints need be satisfied. Yes, `T` must support a `size` member function, though it's worth mentioning that the function might be inherited from a base class. But this member function need not return an integral type. It need not even return a numeric type. For that matter, it need not even return a type for which `operator>` is defined! All it needs to do is return an object of some type `X` such that there is an `operator>` that can be called with an object of type `X` and an `int` (because 10 is of type `int`). The `operator>` need not take a parameter of type `X`, because it could take a parameter of type `Y`, and that would be okay as long as there were an implicit conversion from objects of type `X` to objects of type `Y`!

Similarly, there is no requirement that `T` support `operator!=`, because it would be just as acceptable for `operator!=` to take one object of type `X` and one object of type `Y`. As long as `T` can be converted to `X` and `someNastyWidget`'s type can be converted to `Y`, the call to `operator!=` would be valid.

(As an aside, this analysis doesn't take into account the possibility that `operator&&` could be overloaded, thus changing the meaning of the above expression from a conjunction to something potentially quite different.)

Most people's heads hurt when they first start thinking about implicit interfaces this way, but there's really no need for aspirin. Implicit interfaces are simply made up of a set of valid expressions. The expressions themselves may look complicated, but the constraints they impose are generally straightforward. For example, given the conditional,

```
if (w.size() > 10 && w != someNastyWidget) ...
```

it's hard to say much about the constraints on the functions `size`, `operator>`, `operator&&`, or `operator!=`, but it's easy to identify the constraint on the expression as a whole. The conditional part of an `if` statement must be a boolean expression, so regardless of the exact types involved, whatever "`w.size() > 10 && w != someNastyWidget`" yields, it must be compatible with `bool`. This is part of the implicit interface the template `doProcessing` imposes on its type parameter `T`. The rest of the interface required by `doProcessing` is that calls to the copy constructor, to `normalize`, and to `swap` must be valid for objects of type `T`.

The implicit interfaces imposed on a template's parameters are just as real as the explicit interfaces imposed on a class's objects, and both are checked during compilation. Just as you can't use an object in a way contradictory to the explicit interface its class offers (the code won't compile), you can't try to use an object in a template unless that object supports the implicit interface the template requires (again, the code won't compile).

Things to Remember

- Both classes and templates support interfaces and polymorphism.
- For classes, interfaces are explicit and centered on function signatures. Polymorphism occurs at runtime through virtual functions.
- For template parameters, interfaces are implicit and based on valid expressions. Polymorphism occurs during compilation through template instantiation and function overloading resolution.