

Subroutines and Control Abstraction

8.4.4 Generics in C++, Java, and C#

Though templates were not officially added to C++ until 1990, when the language was almost ten years old, they were envisioned early in its evolution. C# generics, likewise, were planned from the beginning, though they actually didn't appear until the 2.0 release in 2004. By contrast, generics were deliberately omitted from the original version of Java. They were added to Java 5 (also in 2004) in response to strong demand from the user community.

C++ Templates

EXAMPLE 8.69

Generic arbiter class in C++

Figure ©8.13 defines a simple generic class in C++ that we have named an `arbiter`. The purpose of an `arbiter` object is to remember the “best instance” it has seen of some generic parameter class `T`. We have also defined a generic `chooser` class that provides an `operator()` method, allowing it to be called like a function. The intent is that the second generic parameter to `arbiter` should be a subclass of `chooser`, though this is not enforced. Given these definitions we might write

```
class case_sensitive : chooser<string> {
public:
    bool operator()(const string& a, const string& b){return a < b;}
};
...
arbiter<string, case_sensitive> cs_names;           // declare new arbiter
cs_names.consider(new string("Apple"));
cs_names.consider(new string("aardvark"));
cout << *cs_names.best() << "\n";                // prints "Apple"
```

Alternatively, we might define a `case_insensitive` descendant of `chooser`, whereupon we could write

```
template<class T>
class chooser {
public:
    virtual bool operator()(const T& a, const T& b) = 0;
};

template<class T, class C>
class arbiter {
    T* best_so_far;
    C comp;
public:
    arbiter() { best_so_far = 0; }
    void consider(T* t) {
        if (!best_so_far || comp(*t, *best_so_far)) best_so_far = t;
    }
    T* best() {
        return best_so_far;
    }
};
```

Figure 8.13 Generic arbiter in C++.

```
arbiter<string, case_insensitive> ci_names;    // declare new arbiter
ci_names.consider(new string("Apple"));
ci_names.consider(new string("aardvark"));
cout << *ci_names.best() << "\n";           // prints "aardvark"
```

Either way, the C++ compiler will create a new instance of the arbiter template every time we declare an object (e.g., `cs_names`) with a different set of generic arguments. Only when we attempt to use such an object (e.g., by calling `consider`) will it check to see whether the arguments support all the required operations.

Because type checking is delayed until the point of use, there is nothing magic about the `chooser` class. If we neglected to define it, and then left it out of the header of `case_sensitive` (and similarly `case_insensitive`), the code would still compile and run just fine. ■

C++ templates are an extremely powerful facility. Template parameters can include not only types, but also values of ordinary (nongeneric) types, and nested template declarations. Programmers can also define *specialized* templates that provide alternative implementations for certain combinations of arguments. These facilities suffice to implement recursion, giving programmers the ability, at least in principle, to compute arbitrary functions at compile time (in other words, templates are *Turing complete*). An entire branch of software engineering has grown up around so-called *template metaprogramming*, in which templates are used to persuade the C++ compiler to generate custom algorithms for special circumstances [AG90]. As a comparatively simple example, one can write a template that

accepts a generic parameter `int n` and produces a sorting routine for n -element arrays in which all of the loops have been completely unrolled.

As described in Section 8.4.3, C++ allows generic parameters to be *inferred* for generic functions, rather than specified explicitly. To identify the right version of a generic function (from among an arbitrary number of specializations), and to deduce the corresponding generic arguments, the compiler must perform a complicated, potentially recursive pattern-matching operation. This pattern matching is, in fact, quite similar to the type inference of ML-family languages, described in Section ©7.2.4. It can, as noted in the sidebar on page ©169, be cast as *unification*.

Unfortunately, per-use instantiation of templates has two significant drawbacks. First, it tends to result in inscrutable error messages. If we define

EXAMPLE 8.70

Instantiation-time errors in C++ templates

```
class foo {
public:
    bool operator()(const string& a, const unsigned int b)
        // wrong type for second parameter, from arbiter's point of view
        { return a.length() < b; }
};
```

and then say

```
arbiter<string, foo> oops;
...
oops.consider(new string("Apple"));           // line 65 of source
```

one might hope to receive an error message along the lines of “line 65: `foo`’s `operator()` method needs to take two arguments of type `string&`.” Instead the Gnu C++ compiler responds

```
simple_best.cc: In member function ‘void arbiter<T, C>::consider(T*)
[with T = std::string, C = foo]’:
simple_best.cc:65:   instantiated from here
simple_best.cc:21: error: no match for call to ‘(foo)
(std::basic_string<char, std::char_traits<char>,
std::allocator<char> >&, std::basic_string<char,
std::char_traits<char>, std::allocator<char> >&)’
```

(Line 21 is the body of method `consider` in Figure ©8.13.)

Sun’s C++ compiler is equally unhelpful:

```
"simple_best.cc", line 21: Error: Cannot cast from std::basic_string<char,
std::char_traits<char>, std::allocator<char>> to const int.
"simple_best.cc", line 65:   Where: While instantiating
    "arbiter<std::basic_string<char, std::char_traits<char>,
std::allocator<char>>, foo>::consider(std::basic_string<char,
std::char_traits<char>, std::allocator<char>>*)".
"simple_best.cc", line 65:   Where: Instantiated from non-template code.
```

The problem here is fundamental; it's not poor compiler design. Because the language requires that templates be "expanded out" before they are type checked, it is extraordinarily difficult to generate messages without reflecting that expansion. ■

A second drawback of per-use instantiation is a tendency toward "code bloat": it can be difficult, in the presence of separate compilation, to recognize that the same template has been instantiated with the same arguments in separate compilation units. A program compiled in 20 pieces may have 20 copies of a popular template instance.

Java Generics

Generics were deliberately omitted from the original version of Java. Rather than instantiate containers with different generic parameter types, Java programmers followed a convention in which all objects in a container were assumed to be of the standard base class `Object`, from which all other classes are descended. Users of a container could place any type of object inside. When removing an object, however, a *cast* would be needed to reassert the original type. No danger was involved, because objects in Java are self-descriptive, and casts employ run-time checks.

Though dramatically simpler than the use of templates in C++, this programming convention has three significant drawbacks: (1) users of containers must litter their code with casts, which many people find distracting or aesthetically distasteful; (2) errors in the use of a container manifest themselves as `ClassCastExceptions` at run time, rather than as compile-time error messages; (3) the casts incur overhead at run time. Given Java's emphasis on clarity of expression, rather than pure performance, problems (1) and (2) were considered the most serious, and became the subject of a Java Community Process proposal for a language extension in Java 5. The solution adopted is based on the GJ (Generic Java) work of Bracha et al. [BOSW98].

EXAMPLE 8.71

Generic arbiter class in Java

Figure ©8.14 contains a Java 5 version of our arbiter class. It differs from Figure ©8.13 in several important ways. First, Java insists that all instances of a generic be able to share the same code. Among other things, this means that the `Chooser` to be used by a given instance must be specified as a constructor parameter; it cannot be a generic parameter. (We could have used a constructor parameter in C++; in Java it is mandatory.) Second, Java requires that the code for the `arbiter` class be manifestly type-safe, independent of any particular instantiation. We must therefore declare `comp` to be a `Chooser`, so we know that it provides a `better` method. This raises the question: what sort of `Chooser` do we need? That is, what should be the generic parameter in the declaration of `comp` (and of the parameter `c` in the `Arbiter` constructor)?

The most obvious choice (*not* the one adopted in Figure ©8.14) would be `Chooser<T>`. This would allow us to write

```

interface Chooser<T> {
    public boolean better(T a, T b);
}

class Arbiter<T> {
    T bestSoFar;
    Chooser<? super T> comp;

    public Arbiter(Chooser<? super T> c) {
        comp = c;
    }
    public void consider(T t) {
        if(bestSoFar == null || comp.better(t, bestSoFar))bestSoFar = t;
    }
    public T best() {
        return bestSoFar;
    }
}

```

Figure 8.14 Generic arbiter in Java.

```

class CaseSensitive implements Chooser<String> {
    public boolean better(String a, String b) {
        return a.compareTo(b) < 1;
    }
}
...
Arbiter<String> csNames = new Arbiter<String>(new CaseSensitive());
csNames.consider(new String("Apple"));
csNames.consider(new String("aardvark"));
System.out.println(csNames.best());           // prints "Apple" ■

```

EXAMPLE 8.72

Wildcards and bounds on
Java generic parameters

Suppose, however, we were to define

```

class CaseInsensitive implements Chooser<Object> { // note type!
    public boolean better(Object a, Object b) {
        return a.toString().compareToIgnoreCase(b.toString()) < 1;
    }
}

```

Class `Object` defines a `toString` method (usually used for debugging purposes), so this declaration is valid. Moreover since every `String` is an `Object`, we ought to be able to pass any pair of strings to `CaseInsensitive.better` and get a valid response. Unfortunately, `Chooser<Object>` is not acceptable as a match for `Chooser<String>`. If we typed

```
Arbiter<String> ciNames = new Arbiter<String>(new CaseInsensitive());
```

```
interface Chooser {
    public boolean better(Object a, Object b);
}

class Arbiter {
    Object bestSoFar;
    Chooser comp;

    public Arbiter(Chooser c) {
        comp = c;
    }
    public void consider(Object t) {
        if(bestSoFar == null || comp.better(t, bestSoFar))bestSoFar = t;
    }
    public Object best() {
        return bestSoFar;
    }
}
```

Figure 8.15 Arbiter in Java after type erasure. No casts are required in this portion of the code (but see the main text for uses).

the compiler would complain. The fix (as shown in Figure 8.14) is to declare both `comp` and `c` to be of type `<? super T>` instead. This informs the Java compiler that an arbitrary type argument (“?”) is acceptable as the generic parameter of our `Chooser`, so long as that type is an ancestor of `T`.

The `super` keyword specifies a *lower bound* on a type parameter. It is the symmetric opposite of the `extends` keyword, which we used in Example 8.37 to specify an *upper bound*. Together, upper and lower bounds allow us to broaden the set of types that can be used to instantiate generics. As a general rule, we use `extends T` whenever we need to invoke `T` methods; we use `super T` whenever we expect to pass a `T` object as a parameter, but don’t mind if the receiver is willing to accept something more general. Given the bounded declarations used in Figure 8.14, our use of `CaseInsensitive` will compile and run just fine:

```
Arbiter<String> ciNames = new Arbiter<String>(new CaseInsensitive());
ciNames.consider(new String("Apple"));
ciNames.consider(new String("aardvark"));
System.out.println(ciNames.best());           // prints "aardvark" ■
```

Type Erasure

Generics in Java are defined in terms of *type erasure*: the compiler effectively deletes every generic parameter and argument list, replaces every occurrence of a type parameter with `Object`, and inserts casts back to concrete types wherever objects are returned from generic methods. The erased equivalent of Figure 8.14 appears in Figure 8.15. No casts are required in this portion of the code. On any use of `best`, however, the compiler would insert an implicit cast. The statement

EXAMPLE 8.73

Type erasure and implicit casts

```
String winner = csNames.best();
```

will, in effect, be implicitly replaced with

```
String winner = (String) csNames.best();
```

Also, in order to match the `Chooser<String>` interface, our definition of `CaseSensitive` (Example 8.71) will in effect be replaced with

```
class CaseSensitive implements Chooser {
    public boolean better(Object a, Object b) {
        return ((String) a).compareTo((String) b) < 1;
    }
}
```

The advantage of type erasure over the nongeneric version of the code is that the programmer doesn't have to write the casts. In addition, the compiler is able to verify in most cases that the erased code will never generate a `ClassCastException` at run time. The exceptions occur primarily when, for the sake of interoperability with preexisting code, the programmer assigns a generic collection into a nongeneric collection:

EXAMPLE 8.74

Unchecked warnings in
Java 5

```
Arbiter<String> csNames = new Arbiter<String>(new CaseSensitive());
Arbiter alias = csNames;           // nongeneric
alias.consider(new Integer(3));    // unsafe
```

DESIGN & IMPLEMENTATION

Why erasure?

Erasure in Java has several surprising consequences. For one, we can't invoke `new T()`, where `T` is a type parameter: the compiler wouldn't know what kind of object to create. Similarly, Java's *reflection* mechanism, which allows a program to examine and reason about the concrete type of an object at run time, knows nothing about generics: `csNames.getClass().toString()` returns `"class Arbiter"`, not `"class Arbiter<String>"`. Why would the Java designers introduce a mechanism with such significant limitations? The answer is backward compatibility or, more precisely, *migration* compatibility, which requires complete interoperability of old and new code.

More so than most previous languages, Java encourages the assembly of working programs, often on the fly, from components written independently by many different people in many different organizations. The Java designers felt it was critical not only that old (nongeneric) programs be able to run with new (generic) libraries, but also that new (generic) programs be able to run with old (nongeneric) libraries. In addition, they took the position that the Java virtual machine, which interprets Java byte code in the typical implementation, could not be modified. While one can take issue with these goals, once they are accepted erasure becomes a natural solution.

The compiler will issue an “unchecked” warning on the second line of this example, because we have invoked method `consider` on a “raw” (nongeneric) `Arbiter` without explicitly casting the arguments. In this case the warning is clearly warranted: `alias` *shouldn’t* be passed an `Integer`. Other examples can be quite a bit more subtle. It should be emphasized that the warning simply indicates the lack of *static* checking; any type errors that actually occur will still be caught at run time. ■

EXAMPLE 8.75

Java 5 generics and built-in types

Note, by the way, that the use of erasure, and the insistence that every instance of a given generic be able to share the same code, means that type arguments in Java must all be descended from `Object`. While `Arbiter<Integer>` is a perfectly acceptable type, `Arbiter<int>` is not. ■

C# Generics

Though generics were omitted from C# version 1, the language designers always intended to add them, and the .NET Common Language Infrastructure (CLI) was designed from the outset to provide appropriate support. As a result, C# 2.0 was able to employ an implementation based on *reification* rather than erasure. Reification creates a different concrete type every time a generic is instantiated with different arguments. Reified types are visible to the reflection library (`csNames.GetType().ToString()` returns `"Arbiter`1[System.Double]"`), and it is perfectly acceptable to call `new T()` if `T` is a type parameter with a zero-argument constructor (a constraint to this effect is required). Moreover where the Java compiler must generate implicit type casts to satisfy the requirements of the virtual machine (which knows nothing of generics) and to ensure type-safe interaction with legacy code (which might pass a parameter or return a result of an inappropriate type), the C# compiler can be sure that such checks will never be needed, and can therefore leave them out. The result is faster code.

EXAMPLE 8.76

Sharing generic implementations in C#

Of course the C# compiler is free to merge the implementations of any generic instantiations whose code would be the same. Such sharing is significantly easier in C# than it is in C++, because implementations typically employ just-in-time compilation, which delays the generation of machine code until immediately prior to execution, when it’s clear whether an identical instantiation already exists somewhere else in the program. In particular, `MyType<Foo>` and `MyType<Bar>` will share code whenever `Foo` and `Bar` are both classes, because C# employs a reference model for variables of class type. ■

EXAMPLE 8.77

C# generics and built-in types

Like C++, C# allows generic arguments to be value types (built-ins or structs), not just classes. We are free to create an object of class `MyType<int>`; we do not have to “wrap” it as `MyType<Integer>`, the way we would in Java. `MyType<int>` and `MyType<double>` would generally not share code, but both would run significantly faster than `MyType<Integer>` or `MyType<Double>`, because they wouldn’t incur the dynamic memory allocation required to create a wrapper object, the garbage collection required to reclaim it, or the indirection overhead required to access the data inside. ■

Like Java, C# allows only types as generic parameters, and insists that generics be manifestly type-safe, independent of any particular instantiation. It generates


```

public delegate bool Chooser<T>(T a, T b);

class Arbiter<T> {
    T bestSoFar;
    Chooser<T> comp;
    bool initialized;

    public Arbiter(Chooser<T> c) {
        comp = c;
        bestSoFar = default(T);
        initialized = false;
    }
    public void Consider(T t) {
        if (!initialized || comp(t, bestSoFar)) bestSoFar = t;
        initialized = true;
    }
    public T Best() {
        return bestSoFar;
    }
}

```

Figure 8.16 Generic arbiter in C#.

reasonable error messages if we try to instantiate a generic with an argument that doesn't meet the constraints of the corresponding generic parameter, or if we try, inside the generic, to invoke a method that the constraints don't guarantee will be available.

EXAMPLE 8.78

Generic arbiter class in C#

A C# version of our `Arbiter` class appears in Figure ©8.16. One small difference with respect to Figure ©8.14 appears in the `Arbiter` constructor, which must explicitly initialize field `bestSoFar` to `default(T)`. We can leave this out in Java because variables of class type are implicitly initialized to `null`, and type parameters in Java are all classes. In C# `T` might be a built-in or a `struct`, both of which require explicit initialization.

More interesting differences from Figure ©8.14 are the definition of `Chooser` as a *delegate*, rather than an interface, and the lack of lower bounds (uses of the `super` keyword) in parameter and field declarations. These issues are connected. C# allows us to specify an *upper* bound as a type constraint; we did so in the `sort` routine of Example 8.38. There is no direct equivalent, however, for Java's lower bounds. We can work around the problem in the `Arbiter` example by exploiting the fact that `Chooser` has only one method (named `better` in Figure ©8.14).

As described in Section 3.6.3, a C# delegate is a first-class subroutine. The delegate declaration in Figure ©8.16 is roughly analogous to the C declaration

```
typedef _Bool (*Chooser)(T a, T b);
```

(pointer to function of two `T` arguments, returning a Boolean), except that a C# `Chooser` object is a closure, not a pointer: it can refer to a static function, a method

of a particular object (in which case it has access to the object's fields), or an anonymous nested function (in which case it has access, with unlimited extent, to variables in the surrounding scope). In our particular case, defining `Chooser` to be a delegate allows us to pass any appropriate function to the `Arbiter` constructor, without regard to the class inheritance hierarchy. We can declare

```
public static bool CaseSensitive(String a, String b) {
    return String.CompareOrdinal(a, b) < 1;
    // use Unicode order, in which upper-case letters come first
}
public static bool CaseInsensitive(Object a, Object b) {
    return String.Compare(a.ToString(), b.ToString(), false) < 1;
}
```

and then say

```
Arbiter<String> csNames =
    new Arbiter<String>(new Chooser<String>(CaseSensitive));
csNames.Consider("Apple");
csNames.Consider("aardvark");
Console.WriteLine(csNames.Best());           // prints "Apple"

Arbiter<String> ciNames =
    new Arbiter<String>(new Chooser<String>(CaseInsensitive));
ciNames.Consider("Apple");
ciNames.Consider("aardvark");
Console.WriteLine(ciNames.Best());           // prints "aardvark"
```

The compiler is perfectly happy to instantiate `CaseInsensitive` as a `Chooser<String>`, because `Strings` can be passed as `Objects`. ■

✓ CHECK YOUR UNDERSTANDING

68. Why is it difficult to produce high-quality error messages for misuses of C++ templates?
69. What is *template metaprogramming*?
70. Explain the difference between *upper bounds* and *lower bounds* in Java type constraints. Which of these does C# support?
71. What is *type erasure*? Why is it used in Java?
72. Under what circumstances will a Java compiler issue an “unchecked” generic warning?
73. For what two main reasons are C# generics less susceptible to “code bloat” than C++ templates are?

74. Why must fields of generic parameter type be explicitly initialized in C#?
 75. For what two main reasons are C# generics often more efficient than comparable code in Java?
 76. How does a C# *delegate* differ from an interface with a single method (e.g., the C++ *chooser* of Figure ©8.13? How does it differ from a function pointer in C?
-

