11
Multimethods

This chapter defines, discusses, and implements multimethods in the context of C++.

The C++ virtual function mechanism allows dispatching a call depending on the dynamic type of one object. The multimethods feature allows dispatching a function call depending on the types of *multiple* objects. A universally good implementation requires language support, which is the route that languages such as CLOS, ML, Haskell, and Dylan have taken. C++ lacks such support, so its emulation is left to library writers.

This chapter discusses some typical solutions and some generic implementations of each. The solutions feature various trade-offs in terms of speed, flexibility, and dependency management. To describe the technique of dispatching a function call depending on multiple objects, this book uses the terms *multimethod* (borrowed from CLOS) and *multiple dispatch.* A particularization of multiple dispatch for two objects is known as *double dispatch.*

Implementing multimethods is a problem as fascinating as dreaded, one that has stolen lots of hours of good, healthy sleep from designers and programmers.[1]

The topics of this chapter include

- Defining multimethods
- Identifying situations in which the need for multiobject polymorphism appears
- Discussing and implementing three double dispatchers that foster different trade-offs
- Enhancing double-dispatch engines

After reading this chapter, you will have a firm grasp of the typical situations for which multimethods are the way to go. In addition, you will be able to use and extend several robust generic components implementing multimethods, provided by Loki.

This chapter limits discussion to multimethods for two objects (double dispatch). You can use the underlying techniques to extend the number of supported objects to three or more. It is likely, though, that in most situations you can get away with dispatching depending on two objects, and therefore you'll be able to use Loki directly.

[1]If you have trouble implementing multimethods, you can look at this chapter as a sleep aid—which I hope doesn't mean it has an actual soporific effect.

11.1 What Are Multimethods?

In C++, polymorphism essentially means that a given function call can be bound to different implementations, depending on compile-time or runtime contextual issues.

Two types of polymorphism are implemented in C++:

- Compile-time polymorphism, supported by overloading and template functions[2]
- Runtime polymorphism, implemented with virtual functions

Overloading is a simple form of polymorphism that allows multiple functions with the same name to coexist in a scope. If the functions have different parameter lists, the compiler is able to differentiate among them at compile time. Overloading is simple syntactic sugar, a convenient syntactic abbreviation.

Template functions are a static dispatch mechanism. They offer more sophisticated compile-time polymorphism.

Virtual member function calls allow the C++ runtime support, instead of the compiler, to decide which actual function implementation to call. Virtual functions bind a name to a function implementation at runtime. The function called depends on the dynamic type of the object for which you make the virtual call.

Let's now see how these three mechanisms for polymorphism scale to multiple objects. Overloading and template functions scale to multiple objects naturally. Both features allow multiple parameters, and intricate compile-time rules govern function selection.

Unfortunately, virtual functions—the only mechanism that implements runtime polymorphism in C++—are tailored for one object only. Even the call syntax—`obj.Fun` (*arguments*)—gives `obj` a privileged role over *arguments*. (In fact, you can think of `obj` as nothing more than one of `Fun`'s arguments, accessible inside `Fun` as `*this`. The Dylan language, for example, accepts the dot call syntax only as a particular expression of a general function-call mechanism.)

We define *multimethods* or *multiple dispatch* as the mechanism that dispatches a function call to different concrete functions depending on the dynamic types of multiple objects involved in the call. Because we can take compile-time multiobject polymorphism for granted, we only have to implement runtime multiobject polymorphism. Don't be worried; there's a lot left to talk about.

11.2 When Are Multimethods Needed?

Detecting the need for multimethods is simple. You have an operation that manipulates multiple polymorphic objects through pointers or references to their base classes. You would like the behavior of that operation to vary with the dynamic type of more than one of those objects.

Collisions form a typical category of problems best solved with multimethods. For instance, you might write a video game in which moving objects are derived from a `Game`

[2] A more generous view of polymorphism would qualify automatic conversions as the crudest form of polymorphism. They allow, for example, `std::sin` to be called with an `int` although it was written for a `double`. This polymorphism through coercion is only apparent, because the same function call will be issued for both types.

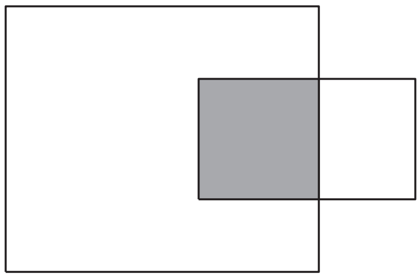

Figure 11.1: Hatching the intersection of two shapes

`Object` abstract class. You would like their collision to behave differently depending on which two types collide: a space ship with an asteroid, a ship with a space station, or an asteroid with a space station.[3]

An example that this chapter uses is marking overlapping areas of drawing objects. Suppose you write a drawing program that allows its users to define rectangles, circles, ellipses, polygons, and other shapes. The basic design is an object-oriented classic: Define an abstract class `Shape` and have all the concrete shapes derive from it; then manipulate a drawing as a collection of pointers to `Shape`.

Now say the client comes and asks for a nice-to-have feature: If two closed shapes intersect, the intersection should be drawn in a way different than each of the two shapes. For instance, the intersection area could be hatched, as in Figure 11.1.

Finding a single algorithm that will hatch any intersection is difficult. For instance, the algorithm that hatches the intersection between an ellipse and a rectangle is very different (and much more complex) from the one that hatches the intersection between two rectangles. Besides, an overly general algorithm (for instance, one that operates at a pixel level) is likely to be highly inefficient, since some intersections (such as rectangle-rectangle) are trivial.

What you need here is a battery of algorithms, each specialized for two shape types, such as rectangle-rectangle, rectangle-polygon, polygon-polygon, ellipse-rectangle, ellipse-polygon, and ellipse-ellipse. At runtime, as the user moves shapes on the screen, you'd like to pick up and fire the appropriate algorithms, which in turn will quickly compute and hatch the overlapping areas.

Because you manipulate all drawing objects as pointers to `Shape`, you don't have the type information necessary to select the suitable algorithm. You have to start from pointers to `Shape` only. Because you have two objects involved, simple virtual functions cannot solve this problem. You have to use double dispatch.

11.3 Double Switch-on-Type: Brute Force

The most straightforward approach to a double dispatch implementation is to implement a double switch-on-type. You try to dynamic cast the first object against each of the possible left-hand types in succession. For each branch, you do the same with the second argument.

[3]This example and names were borrowed from Scott Meyers's *More Effective C++* (1996a), Item 31.

When you have discovered the types of both objects, you know what function to call. The code looks like this:

```
// various intersection algorithms
void DoHatchArea1(Rectangle&, Rectangle&);
void DoHatchArea2(Rectangle&, Ellipse&);
void DoHatchArea3(Rectangle&, Poly&);
void DoHatchArea4(Ellipse&, Poly&);
void DoHatchArea5(Ellipse&, Ellipse&);
void DoHatchArea6(Poly&, Poly&);

void DoubleDispatch(Shape& lhs, Shape& rhs)
{
   if (Rectangle* p1 = dynamic_cast<Rectangle*>(&lhs))
   {
      if (Rectangle* p2 = dynamic_cast<Rectangle*>(&rhs))
         DoHatchArea1(*p1, *p2);
      else if (Ellipse p2 = dynamic_cast<Ellipse*>(&rhs))
         DoHatchArea2(*p1, *p2);
      else if (Poly p2 = dynamic_cast<Poly*>(&rhs))
         DoHatchArea3(*p1, *p2);
      else
         Error("Undefined Intersection");
   }
   else if (Ellipse* p1 = dynamic_cast<Ellipse*>(&lhs))
   {
      if (Rectangle* p2 = dynamic_cast<Rectangle*>(&rhs))
         DoHatchArea2(*p2, *p1);
      else if (Ellipse* p2 = dynamic_cast<Ellipse*>(&rhs))
         DoHatchArea5(*p1, *p2);
      else if (Poly* p2 = dynamic_cast<Poly*>(&rhs))
         DoHatchArea4(*p1, *p2);
      else
         Error("Undefined Intersection");
   }
   else if (Poly* p1 = dynamic_cast<Poly*>(&lhs))
   {
      if (Rectangle* p2 = dynamic_cast<Rectangle*>(&rhs))
         DoHatchArea2(*p2, *p1);
      else if (Ellipse* p2 = dynamic_cast<Ellipse*>(&rhs))
         DoHatchArea4(*p2, *p1);
      else if (Poly* p2 = dynamic_cast<Poly*>(&rhs))
         DoHatchArea6(*p1, *p2);
      else
         Error("Undefined Intersection");
   }
   else
   {
      Error("Undefined Intersection");
   }
}
```

Whew! It's been quite a few lines. As you can see, the brute-force approach asks you to write a lot of (allegedly trivial) code. You can count on any dependable C++ programmer to put together the appropriate net of `if` statements. In addition, the solution has the advantage of being fast if the number of possible classes is not too high. From a speed perspective, `DoubleDispatch` implements a linear search in the set of possible types. Because the search is unrolled—a succession of `if-else` statements as opposed to a loop—the speed is very good for small sets.

One problem with the brute-force approach is sheer code size, which makes the code unmaintainable as the number of classes grows. The code just given deals only with three classes, yet it's already of considerable size. The size grows exponentially as you add more classes. Imagine how the code of `DoubleDispatch` would look for a hierarchy of 20 classes!

Another problem is that `DoubleDispatch` is a dependency bottleneck—its implementation must know of the existence of all classes in a hierarchy. It is best to keep the dependency net as loose as possible, and `DoubleDispatch` is a dependency hog.

The third problem with `DoubleDispatch` is that the order of the `if` statements matters. This is a very subtle and dangerous problem. Imagine, for instance, you derive class `RoundedRectangle` (a rectangle with rounded corners) from `Rectangle`. You then edit `DoubleDispatch` and insert the additional `if` statement at the end of each `if-else` statement, right before the `Error` call. You have just introduced a bug.

The reason is that if you pass `DoubleDispatch` a pointer to a `RoundedRectangle`, the `dynamic_cast<Rectangle*>` succeeds. Because that test is before the test for `dynamic_cast<RoundedRectangle*>`, the first test will "eat" both `Rectangles` and `Rounded Rectangles`. The second test will never get a chance. Most compilers don't warn about this.

A candidate solution would be to change the tests as follows:

```
void DoubleDispatch(Shape& lhs, Shape& rhs)
{
   if (typeid(lhs) == typeid(Rectangle))
   {
      Rectangle* p1 = dynamic_cast<Rectangle*>(&lhs);
      ...
   }
   else ...
}
```

The tests are now for the exact type instead of the exact or derived type. The `typeid` comparison shown in this code fails if `lhs` is a `RoundedRectangle`, so the tests continue. Ultimately, the test against `typeid(RoundedRectangle)` succeeds.

Alas, this fixes one aspect but breaks another: `DoubleDispatch` is too rigid now. If you didn't add support for a type in `DoubleDispatch`, you would like `DoubleDispatch` to fire on the closest base type. This is what you'd normally expect when using inheritance—by default, derived objects do what base objects do unless you override some behavior. The problem is that the `typeid`-based implementation of `DoubleDispatch` fails to preserve this property. The rule of thumb that results from this fact is that you must still use `dynamic_cast` in `DoubleDispatch` and "sort" the `if` tests so that the most derived classes are tried first.

This adds two more disadvantages to the brute-force implementation of multimethods.

First, the dependency between DoubleDispatch and the Shape hierarchy deepens—DoubleDispatch must know about not only classes but also the inheritance relationships between classes. Second, maintaining the appropriate ordering of dynamic casts puts a supplemental burden on the shoulders of the maintainer.

11.4 The Brute-Force Approach Automated

Because in some situations the speed of the brute-force approach can be unbeatable, it's worth paying attention to implementing such a dispatcher. Here's where typelists can be of help.

Recall from Chapter 3 that Loki defines a typelist facility—a collection of structures and compile-time algorithms that allow you to manipulate collections of types. A brute-force implementation of multimethods can use a client-provided typelist that specifies the classes in the hierarchy (in our example, Rectangle, Poly, Ellipse, etc.). Then a recursive template can generate the sequence of if-else statements.

In the general case, we can dispatch on different collections of types, so the typelist for the left-hand operand and the one for the right-hand operand can be different.

Let's try outlining a StaticDispatcher class template that performs the type deduction algorithm, then fires a function in another class. Explanations follow the code.

```
template
<
   class Executor,
   class BaseLhs,
   class TypesLhs,
   class BaseRhs = BaseLhs,
   class TypesRhs = TypesLhs
   typename ResultType = void
>
class StaticDispatcher
{
   typedef typename TypesLhs::Head Head;
   typedef typename TypesLhs::Tail Tail;
public:
   static ResultType Go(BaseLhs& lhs, BaseRhs& rhs,
      Executor exec)
   {
      if (Head* p1 = dynamic_cast<Head*>(&lhs))
      {
         return StaticDispatcher<Executor, BaseLhs,
            NullType, BaseRhs, TypesRhs>::DispatchRhs(
               *p1, rhs, exec);
      }
      else
      {
         return StaticDispatcher<Executor, BaseLhs,
            Tail, BaseRhs, TypesRhs>::Go(
               lhs, rhs, exec);
      }
   }
```

```
      ...
   };
```

If you are familiar with typelists, the workings of StaticDispatcher are seen to be quite simple. StaticDispatcher has surprisingly little code for what it does.

StaticDispatcher has six template parameters. Executor is the type of the object that does the actual processing—in our example, hatching the intersection area. We'll discuss what Executor looks like a bit later.

BaseLhs and BaseRhs are the base types of the arguments on the left-hand side and the right-hand side, respectively. TypesLhs and TypesRhs are typelists containing the possible derived types for the two arguments. The default values of BaseRhs and TypesRhs foster a dispatcher for types in the same class hierarchy, as is the case with the drawing program example.

ResultType is the type of the result of the double-dispatch operation. In the general case, the dispatched function can return an arbitrary type. StaticDispatcher supports this dimension of genericity and forwards the result to the caller.

StaticDispatcher::Go tries a dynamic cast to the first type found in the TypesLhs typelist, against the address of lhs. If the dynamic cast fails, Go delegates to the remainder (tail) of TypesLhs in a recursive call to itself. (This is not a true recursive call, because each time we have a different instantiation of StaticDispatcher.)

The net effect is that Go performs a suite of if-else statements that apply dynamic_cast to each type in the typelist. When a match is found, Go invokes DispatchRhs. DispatchRhs does the second and last step of the type deduction—finding the dynamic type of rhs.

```
   template <...>
   class StaticDispatcher
   {
      ... as above ...
      template <class SomeLhs>
      static ResultType DispatchRhs(SomeLhs& lhs, BaseRhs& rhs,
         Executor exec)
      {
         typedef typename TypesRhs::Head Head;
         typedef typename TypesRhs::Tail Tail;

         if (Head* p2 = dynamic_cast<Head*>(&rhs))
         {
            return exec.Fire(lhs, *p2);
         }
         else
         {
            return StaticDispatcher<Executor, SomeLhs,
               NullType, BaseRhs, Tail>::DispatchRhs(
                    lhs, rhs, exec);
         }
      }
   };
```

DispatchRhs performs the same algorithm for rhs as Go applied for lhs. In addition, when the dynamic cast on rhs succeeds, DispatchRhs calls Executor::Fire, passing it the two discovered types. Again, the code that the compiler generates is a suite of if-else statements. Interestingly, the compiler generates one such suite of if-else statements for *each* type in TypesLhs. Effectively, StaticDispatcher manages to generate an exponential amount of code with two typelists and a fixed codebase. This is an asset, but also a potential danger—too much code can hurt compile times, program size, and execution time all together.

To treat the limit conditions that stop the compile-time recursion, we need to specialize StaticDispatcher for two cases: The type of lhs is not found in TypesLhs, and the type of rhs is not found in TypesRhs.

The first case (error on lhs) appears when you invoke Go on a StaticDispatcher with NullType as TypesLhs. This is the sign that the search depleted TypesLhs. (Recall from Chapter 3 that NullType is used to signal the last element in any typelist.)

```
template
<
   class Executor,
   class BaseLhs,
   class BaseRhs,
   class TypesRhs,
   typename ResultType
>
class StaticDispatcher<Executor, BaseLhs, NullType,
   BaseRhs, TypesRhs, ResultType>
{
   static void Go(BaseLhs& lhs, BaseRhs& rhs, Executor exec)
   {
      exec.OnError(lhs, rhs);
   }
};
```

Error handling is elegantly delegated to the Executor class, as you will see in the discussion on Executor later.

The second case (error on rhs) appears when you invoke DispatchRhs on a Static Dispatcher with NullType as TypesRhs. Hence the following specialization:

```
template
<
   class Executor,
   class BaseLhs,
   class TypesLhs,
   class BaseRhs,
   class TypesRhs,
   typename ResultType
>
class StaticDispatcher<Executor, BaseLhs, TypesLhs,
   BaseRhs, NullType, ResultType>
{
public:
      static void DispatchRhs(BaseLhs& lhs, BaseRhs& rhs,
         Executor& exec)
      {
         exec.OnError(lhs, rhs);
      }
   };
```

It is time now to discuss what Executor has to implement to take advantage of the double-dispatch engine we have just defined.

StaticDispatcher deals only with type discovery. After finding the right types and objects, it passes them to a call of Executor::Fire. To differentiate these calls, Executor must implement several overloads of Fire. For example, the Executor class for hatching shape intersections is as follows:

```
   class HatchingExecutor
   {
   public:
      // Various intersection algorithms
      void Fire(Rectangle&, Rectangle&);
      void Fire(Rectangle&, Ellipse&);
      void Fire(Rectangle&, Poly&);
      void Fire(Ellipse&, Poly&);
      void Fire(Ellipse&, Ellipse&);
      void Fire(Poly&, Poly&);

      // Error handling routine
      void OnError(Shape&, Shape&);
   };
```

You use HatchingExecutor with StaticDispatcher as shown in the following code:

```
   typedef StaticDispatcher<HatchingExecutor, Shape,
      TYPELIST_3(Rectangle, Ellipse, Poly)> Dispatcher;
   Shape *p1 = ...;
   Shape *p2 = ...;
   HatchingExecutor exec;
   Dispatcher::Go(*p1, *p2, exec);
```

This code invokes the appropriate Fire overload in the HatchingExecutor class. You can see the StaticDispatcher class template as a mechanism that achieves dynamic overloading—it defers overloading rules to runtime. This makes StaticDispatcher remarkably easy to use. You just implement HatchingExecutor with the overloading rules in mind, and then you use StaticDispatcher as a black box that does the magic of applying overloading rules at runtime.

As a nice side effect, StaticDispatcher will unveil any overloading ambiguities at compile time. For instance, assume you don't declare HatchingExecutor::Fire(Ellipse&, Poly&). Instead, you declare HatchingExecutor::Fire(Ellipse&,Shape&) and Hatching Executor::Fire(Shape&, Poly&). Calling Hatching Executor::Fire with an Ellipse and a Poly would result in an ambiguity—both functions compete to handle the call.

Remarkably, StaticDispatcher signals the same error for you and with the same level of detail. StaticDispatcher is a tool that's very consistent with the existing C++ overloading rules.

What happens in the case of a runtime error—for instance, if you pass a Circle as one of the arguments of StaticDispatcher::Go? As hinted earlier, StaticDispatcher handles border cases by simply calling Executor::OnError with the original (not casted) lhs and rhs. This means that, in our example, HatchingExecutor::OnError (Shape&, Shape&) is the error handling routine. You can use this routine to do whatever you find appropriate—when it's called, it means that StaticDispatcher gave up on finding the dynamic types.

As discussed in the previous section, inheritance adds additional problems to a brute-force dispatcher. That is, the following instantiation of StaticDispatcher has a bug:

```
typedef StaticDispatcher
<
   SomeExecutor,
   Shape,
   TYPELIST_4(Rectangle, Ellipse, Poly, RoundedRectangle)
>
MyDispatcher;
```

If you pass a RoundedRectangle to MyDispatcher, it will be considered a Rectangle. The dynamic_cast<Rectangle*> succeeds on a pointer to a RoundedRectangle, and because the dynamic_cast<RoundedRectangle*> is lower down on the food chain, it will never be given a chance. The correct instantiation is

```
typedef StaticDispatcher
<
   SomeExecutor,
   Shape,
   TYPELIST_4(RoundedRectangle, Ellipse, Poly, Rectangle)
>
Dispatcher;
```

The general rule is to put the most derived types at the front of the typelist.

It would be nice if this transformation could be applied automatically, and typelists do support that. We have a means to detect inheritance at compile time (Chapter 2), and typelists can be sorted. This led to the DerivedToFront compile-time algorithm in Chapter 3.

All we have to do to take advantage of automatic sorting is to modify the implementation of StaticDispatcher as follows:

```
template <...>
class StaticDispatcher
{
   typedef typename DerivedToFront<
      typename TypesLhs::Head>::Result Head;
   typedef typename DerivedToFront<
      typename TypesLhs::Tail>::Result Tail;
public:
   ... as above ...
};
```

After all this handy automation, don't forget that all we obtained is the code generation part. The dependency problems are still with us. Although it makes it very easy to implement brute-force multimethods, `StaticDispatcher` still has a dependency on all the types in the hierarchy. Its advantages are speed (if there are not too many types in the hierarchy) and nonintrusiveness—you don't have to modify a hierarchy to use `StaticDispatcher` with it.

11.5 Symmetry with the Brute-Force Dispatcher

When you hatch the intersection between two shapes, you might want to do it differently if you have a rectangle covering an ellipse than if you have an ellipse covering a rectangle. Or, on the contrary, you might need to hatch the intersection area the same way when an ellipse and a rectangle intersect, no matter which covers which. In the latter case, you need a *symmetric* multimethod—a multimethod that is insensitive to the order in which you pass its arguments.

Symmetry applies only when the two parameter types are identical (in our case, `BaseLhs` is the same as `BaseRhs`, and `LhsTypes` is the same as `RhsTypes`).

The brute-force `StaticDispatcher` defined previously is asymmetric; that is, it doesn't offer any built-in support for symmetric multimethods. For example, assume you define the following classes:

```
class HatchingExecutor
{
public:
   void Fire(Rectangle&, Rectangle&);
   void Fire(Rectangle&, Ellipse&);
   ...
   // Error handler
   void OnError(Shape&, Shape&);
};

typedef StaticDispatcher
<
   HatchingExecutor,
   Shape,
   TYPELIST_3(Rectangle, Ellipse, Poly)
>
HatchingDispatcher;
```

The `HatchingDispatcher` does not fire when passed an `Ellipse` as the left-hand parameter and a `Rectangle` as the right-hand parameter. Even though from your `HatchingExecutor`'s viewpoint it doesn't matter who's first and who's second, `HatchingDispatcher` will insist that you pass objects in a certain order.

Fixing the symmetry in the client code is possible by reversing arguments and forwarding from one overload to another:

```
class HatchingExecutor
{
```

```
public:
   void Fire(Rectangle&, Ellipse&);
   // Symmetry assurance
   void Fire(Ellipse& lhs, Rectangle& rhs)
   {
      // Forward to Fire(Rectangle&, Ellipse&)
      //  by switching the order of arguments
      Fire(rhs, lhs);
   }
   ...
};
```

These little forwarding functions are hard to maintain. Ideally, `StaticDispatcher` would provide itself optional support for symmetry through an additional `bool` template parameter, which is worth looking into.

The need is to have `StaticDispatcher` reverse the order of arguments when invoking the callback, for certain cases. What are those cases? Let's analyze the previous example. Expanding the template argument lists from their default values, we get the following instantiation:

```
typedef StaticDispatcher
<
   HatchingExecutor,
   Shape,
   TYPELIST_2(Rectangle, Ellipse, Poly),  // TypesLhs
   Shape,
   TYPELIST_2(Rectangle, Ellipse, Poly),  // TypesRhs
   void
>
HatchingDispatcher;
```

An algorithm for selecting parameter pairs for a symmetric dispatcher can be as follows: Combine the first type in the first typelist (`TypesLhs`) with each type in the second typelist (`TypesRhs`). This gives three combinations: `Rectangle-Rectangle`, `Rectangle-Ellipse`, and `Rectangle-Poly`. Next, combine the second type in `Types Lhs` (`Ellipse`) with types in `TypesRhs`. However, because the first combination (`Rectangle-Ellipse`) has already been made in the first step, this time start with the second element in `Types Rhs`. This step yields `Ellipse-Ellipse` and `Ellipse-Poly`. The same reasoning applies to the next step: `Poly` in `TypesLhs` must be combined only with types starting with the third one in `TypesRhs`. This gives only one combination, `Poly-Poly`, and the algorithm stops here.

Following this algorithm, you implement only the functions for the selected combination, as follows:

```
class HatchingExecutor
{
public:
   void Fire(Rectangle&, Rectangle&);
   void Fire(Rectangle&, Ellipse&);
   void Fire(Rectangle&, Poly&);
   void Fire(Ellipse&, Ellipse&);
   void Fire(Ellipse&, Poly&);
   void Fire(Poly&, Poly&);
   // Error handler
   void OnError(Shape&, Shape&);
};
```

StaticDispatcher must detect all by itself the combinations that were eliminated by the algorithm just discussed, namely Ellipse-Rectangle, Poly-Rectangle, and Poly-Ellipse. For these three combinations, StaticDispatcher must reverse the arguments. For all others, StaticDispatcher forwards the call just as it did before.

What's the Boolean condition that determines whether or not argument swapping is needed? The algorithm selects the types in TL2 only at indices *greater than or equal to* the index of the type in TL1. Therefore, the condition is as follows:

> For two types T and U, if the index of U in TypesRhs is less than the index of T in TypesLhs, then the arguments must be swapped.

For example, say T is Ellipse and U is Rectangle. Then T's index in TypesLhs is 1 and U's index in TypesRhs is 0. Consequently, Ellipse and Rectangle must be swapped before invoking Executor::Fire, which is correct.

The typelist facility already provides the IndexOf compile-time algorithm that returns the position of a type in a typelist. We can then write the swapping condition easily.

First, we must add a new template parameter that says whether the dispatcher is symmetric. Then, we add a simple little traits class template InvocationTraits that either swaps the arguments or does not swap them when calling the Executor::Fire member function. Here is the relevant excerpt.

```
template
<
   class Executor,
   bool symmetric,
   class BaseLhs,
   class TypesLhs,
   class BaseRhs = BaseLhs,
   class TypesRhs = TypesLhs,
   typename ResultType = void
>
class StaticDispatcher
{
   template <bool swapArgs, class SomeLhs, class SomeRhs>
   struct InvocationTraits
   {
      static void DoDispatch(SomeLhs& lhs, SomeRhs& rhs,
         Executor& exec)
      {
         exec.Fire(lhs, rhs);
      }
   };
   template <class SomeLhs, class SomeRhs>
   struct InvocationTraits<True, SomeLhs, SomeRhs>
   {
         static void DoDispatch(SomeLhs& lhs, SomeRhs& rhs,
            Executor& exec)
         {
            exec.Fire(rhs, lhs); // swap arguments
         }
      }
   public:
      static void DispatchRhs(BaseLhs& lhs, BaseRhs& rhs,
         Executor exec)
      {
         if (Head* p2 = dynamic_cast<Head*>(&rhs))
         {
            enum { swapArgs = symmetric &&
               IndexOf<Head, TypesRhs>::result <
               IndexOf<BaseLhs, TypesLhs>::result };
            typedef InvocationTraits<swapArgs, BaseLhs, Head>
               CallTraits;
            return CallTraits::DoDispatch(lhs, *p2);
         }
         else
         {
            return StaticDispatcher<Executor, BaseLhs,
               NullType, BaseRhs, Tail>::DispatchRhs(
                  lhs, rhs, exec);
         }
      }
   };
```

Support for symmetry adds some complexity to StaticDispatcher, but it certainly makes things much easier for StaticDispatcher's user.

11.6 The Logarithmic Double Dispatcher

If you want to avoid the heavy dependencies accompanying the brute-force solution, you must look into a more dynamic approach. Instead of generating code at compile time, you must keep a runtime structure and use runtime algorithms that help in dynamically dispatching function calls depending on types.

RTTI (runtime type information) can be of further help here because it provides not only dynamic_cast and type identification, but also a runtime ordering of types, through the before member function of std::type_info. What before offers is an ordering relationship on all types in a program. We can use this ordering relationship for fast searches of types.

The implementation here is similar to the one found in Item 31 of Scott Meyers' *More Effective* C++ (1996a), with some improvements: the casting step when invoking a handler is automated, and the implementation herein aims at being generic.

We will avail ourselves of the OrderedTypeInfo class, described in Chapter 2. OrderedTypeInfo is a wrapper providing exactly the same functionality as std::type_info. In addition, OrderedTypeInfo provides value semantics and a caveat-free less-than operator. You can thus store OrderedTypeInfo objects in standard containers, which is of interest to this chapter.

Meyers's approach was simple: For each pair of std::type_info objects you want to dispatch upon, you register a pointer to a function with the double dispatcher. The double dispatcher stores the information in a std::map. At runtime, when invoked with two unknown objects, the double dispatcher performs a fast search (logarithmic time) for type discovery, and if it finds an entry, fires the appropriate pointer to a function.

Let's define the structure of a generic engine operating on these principles. We must templatize the engine with the base types of the two arguments (left-hand side and right-hand side). We call this engine BasicDispatcher, because we will use it as the base device for several more advanced double dispatchers.

```
template <class BaseLhs, class BaseRhs = BaseLhs,
   typename ResultType = void>
class BasicDispatcher
{
   typedef std::pair<OrderedTypeInfo, OrderedTypeInfo>
      KeyType;
   typedef ResultType (*CallbackType)(BaseLhs&, BaseRhs&);
   typedef CallbackType MappedType;
   typedef std::map<KeyType, MappedType> MapType;
   MapType callbackMap_;
public:
   ...
};
```

The key type in the map is a std::pair of two OrderedTypeInfo objects. The std::pair class supports ordering, so we don't have to provide a custom ordering functor.

BasicDispatcher can be more general if we templatize the callback type. In general, the callback does not have to be a function. It can be, for example, a functor (refer to the introduction of Chapter 5 for a discussion of functors). BasicDispatcher can accommodate functors by transforming its inner CallbackType type definition into a template parameter.

An important improvement is to change the std::map type to a more efficient structure. Matt Austern (2000) explains that the standard associative containers have a slightly narrower area of applicability than one might think. In particular, a sorted vector in combination with binary search algorithms (such as std::lower_bound) might perform much better, in both space and time, than an associative container. This happens when the number of accesses is much larger than the number of insertions. So we should take a close look at the typical usage pattern of a double-dispatcher object.

Most often, a double dispatcher is a write-once, read-many type of structure. Typically, a program sets the callbacks once and then uses the dispatcher many, many times. This is in keeping with the virtual functions mechanism, which double dispatchers extend. You decide which functions are virtual and which are not, at compile time.

It seems as if we're better off with a sorted vector. The disadvantages of a sorted vector are linear-time insertions and linear-time deletions, and a double dispatcher is not typically concerned about the speed of either. In exchange, a vector offers about twice the lookup speed and a much smaller working set, so it is definitely a better choice for a double dispatcher.

Loki saves the trouble of maintaining a sorted vector by hand by defining an Assoc-

Vector class template. AssocVector is a drop-in replacement for std::map (it supports the same set of member functions), implemented on top of std::vector. Assoc Vector differs from a map in the behavior of its erase functions (AssocVector:: erase invalidates all iterators into the object) and in the complexity guarantees of insert and erase (linear as opposed to constant). Because of the high degree of compatibility of AssocVector with std::map, we'll continue to use the term *map* to describe the data structure held by the double dispatcher.

Here is the revised definition of BasicDispatcher:

```
template
<
   class BaseLhs,
   class BaseRhs = BaseLhs,
   typename ResultType = void,
   typename CallbackType = ResultType (*)(BaseLhs&, BaseRhs&)
>
class BasicDispatcher
{
   typedef std::pair<TypeInfo, TypeInfo>
      KeyType;
   typedef CallbackType MappedType;
   typedef AssocVector<KeyType, MappedType> MapType;
   MapType callbackMap_;
public:
   ...
};
```

The registration function is easy to define. This is all we need:

```
template <...>
class BasicDispatcher
{
   ... as above ...
   template <class SomeLhs, SomeRhs>
   void Add(CallbackType fun)
   {
      const KeyType key(typeid(SomeLhs), typeid(SomeRhs));
      callbackMap_[key] = fun;
   }
};
```

The types SomeLhs and SomeRhs are the concrete types for which you need to dispatch the call. Just like std::map, AssocVector overloads operator[] to find a key's corresponding mapped type. If the entry is missing, a new element is inserted. Then operator[] returns a reference to that new or found element, and Add assigns fun to it.

The following is an example of using Add:

```
typedef BasicDispatcher<Shape> Dispatcher;
// Hatches the intersection between a rectangle and a polygon
void HatchRectanglePoly(Shape& lhs, Shape& rhs)
{
   Rectangle& rc = dynamic_cast<Rectangle&>(lhs);
```

```
      Poly& pl = dynamic_cast<Poly&>(rhs);
      ... use rc and pl ...
   }
   ...
   Dispatcher disp;
   disp.Add<Rectangle, Poly>(HatchRectanglePoly);
```

The member function that does the search and invocation is simple:

```
   template <...>
   class BasicDispatcher
   {
      ... as above ...
      ResultType Go(BaseLhs& lhs, BaseRhs& rhs)
      {
         MapType::iterator i = callbackMap_.find(
            KeyType(typeid(lhs), typeid(rhs));
         if (i == callbackMap_.end())
         {
            throw std::runtime_error("Function not found");
         }
         return (i->second)(lhs, rhs);
      }
   };
```

11.6.1 The Logarithmic Dispatcher and Inheritance

BasicDispatcher does not work correctly with inheritance. If you register only HatchRectanglePoly(Shape& lhs,Shape&rhs) with BasicDispatcher, you get proper dispatching only for objects of type Rectangle and Poly—nothing else. If, for instance, you pass references to a RoundedRectangle and a Poly to BasicDispatcher::Go, BasicDispatcher will reject the call.

The behavior of BasicDispatcher is not in keeping with inheritance rules, according to which derived types must by default act like their base types. It would be nice if BasicDispatcher accepted calls with objects of derived classes, as long as these calls were unambiguous per C++'s overloading rules.

There are quite a few things you can do to correct this problem, but to date there is no complete solution. You must be careful to register all the pairs of types with BasicDispatcher.[4]

11.6.2 The Logarithmic Dispatcher and Casts

BasicDispatcher is usable, but not quite satisfactory. Although you register a function that handles the intersection between a Rectangle and a Poly, that function must accept arguments of the base type, Shape&. It is awkward and error prone to ask client code (HatchRectanglePoly's implementation) to cast the Shape references back to the correct types.

On the other hand, the callback map cannot hold a different function or functor type for

[4]I am convinced there is a solution to the inheritance problem. But, alas, writers of books have deadlines, too.

each element, so we must stick to a uniform representation. Item 31 in *More Effective C++* (Meyers 1996a) discusses this issue, too. No function-pointer-to-function-pointer cast helps because once you exit `FnDoubleDispatcher::Add`, you've lost the static type information, so you don't know what to cast to. (If this sounds confusing, try spinning some code and you'll immediately figure it out.)

We will implement a solution to the casting problem in the context of simple callback functions (not functors). That is, the `CallbackType` template argument is a pointer to a function.

An idea that could help is using a *trampoline function,* also known as a *thunk.* Trampoline functions are small functions that perform little adjustments before calling other functions. They are commonly used by C++ compiler writers to implement features such as covariant return types and pointer adjustments for multiple inheritance.

We can use a trampoline function that performs the appropriate cast and then calls a function of the proper signature, thus making life much easier for the client. The problem, however, is that `callbackMap_` must now store *two* pointers to functions: one to the pointer provided by the client, and one to the pointer to the trampoline function. This is worrisome in terms of speed. Instead of an indirect call through a pointer, we have two. In addition, the implementation becomes more complicated.

An interesting bit of wizardry saves the day. A template can accept a pointer to a function as a nontype template parameter. (Most often in this book, nontype template parameters are integral values.) A template is allowed to accept pointers to global objects, including functions, as nontype template parameters. The only condition is that the function whose address is used as a template argument must have external linkage. You can easily transform static functions in functions with external linkage by removing `static` and putting them in unnamed namespaces. For example, what you would write as

```
static void Fun();
```

in pre-`namespace` C++, you can write using an anonymous namespace as

```
namespace
{
   void Fun();
}
```

Using a pointer to a function as a nontype template argument means that we don't need to store it in the map anymore. This essential aspect needs thorough understanding. The reason we don't need to store the pointer to a function is that the compiler has static knowledge about it. Thus, the compiler can hardcode the function address in the trampoline code.

We implement this idea in a new class that uses `BasicDispatcher` as its back end. The new class, `FnDispatcher`, is tuned for dispatching to functions only—not to functors. `FnDispatcher` aggregates `BasicDispatcher` privately and provides appropriate forwarding functions.

The `FnDispatcher::Add` template function accepts three template parameters. Two represent the left-hand-side and the right-hand-side types for which the dispatch is registered

(ConcreteLhs and ConcreteRhs). The third template parameter (callback) is the pointer to a function. The added FnDispatcher::Add overloads the template Add with only two template parameters, defined earlier.

```
template <class BaseLhs, class BaseRhs = BaseLhs,
   ResultType = void>
class FnDispatcher
{
   BaseDispatcher<BaseLhs, BaseRhs, ResultType> backEnd_;
   ...
public:
   template <class ConcreteLhs, class ConcreteRhs,
      ResultType (*callback)(ConcreteLhs&, ConcreteRhs&)>
   void Add()
   {
      struct Local // see Chapter 2
      {
         static ResultType Trampoline(BaseLhs& lhs, BaseRhs& rhs)
         {
            return callback(
               dynamic_cast<ConcreteLhs&>(lhs),
               dynamic_cast<ConcreteRhs&>(rhs));
         }
      };
      return backEnd_.Add<ConcreteLhs, ConcreteRhs>(
         &Local::Trampoline);
   }
};
```

Using a local structure, we define the trampoline right inside Add. The trampoline casts the arguments to the right types and then forwards to callback. Then, the Add function uses backEnd_'s Add function (defined by BaseDispatcher) to add the trampoline to callbackMap_.

As far as speed is concerned, the trampoline does not incur additional overhead. Although it looks like an indirect call, the call to callback is not. As said before, the compiler hardwires callback's address right into Trampoline so there is no second indirection. A clever compiler can even inline the call to callback if possible.

Using the newly defined Add function is simple:

```
typedef FnDispatcher<Shape> Dispatcher;

// Possibly in an unnamed namespace
void HatchRectanglePoly(Rectangle& lhs, Poly& rhs)
{
   ...
}

Dispatcher disp;
disp.Add<Rectangle, Poly, Hatch>();
```

Because of its `Add` member function, `FnDispatcher` is easy to use. `FnDispatcher` also exposes an `Add` function similar to the one defined by `BaseDispatcher`, so you still can use this function if you need.[5]

11.7 `FnDispatcher` and Symmetry

Because of `FnDispatcher`'s dynamism, adding support for symmetry is much easier than it was with the static `StaticDispatcher`.

All we have to do to support symmetry is to register two trampolines: one that calls the executor in normal order, and one that swaps the parameters before calling. We add a new template parameter to `Add`, as shown.

```
template <class BaseLhs, class BaseRhs = BaseLhs,
   typename ResultType = void>
class FnDispatcher
{
   ...
   template <class ConcreteLhs, class ConcreteRhs,
      ResultType (*callback)(ConcreteLhs&, ConcreteRhs&),
      bool symmetric>
   bool Add()
   {
      struct Local
      {
         ... Trampoline as before ...
         static void TrampolineR(BaseRhs& rhs, BaseLhs& lhs)
         {
            return Trampoline(lhs, rhs);
         }
      };
      Add<ConcreteLhs, ConcreteRhs>(&Local::Trampoline);
      if (symmetric)
      {
         Add<ConcreteRhs, ConcreteLhs>(&Local::TrampolineR);
      }
   }
};
```

Symmetry with `FnDispatcher` has function-level granularity—for each function you register, you can decide whether you want symmetric dispatching or not.

11.8 Double Dispatch to Functors

As described earlier, the trampoline trick works nicely with pointers to nonstatic functions. Anonymous namespaces provide a clean way to replace static functions with nonstatic functions that are not visible outside the current compilation unit.

[5]One case in which you cannot use `FnDispatcher::Add` is when you need to register dynamically loaded functions. Even in this case, however, you can make slight changes to your design so you can take advantage of trampolines.

Sometimes, however, you need your callback objects (the `CallbackType` template parameter of `BasicDispatcher`) to be more substantial than simple pointers to functions. For instance, you might want each callback to hold some state, and functions cannot hold much state (only static variables). Consequently, you need to register *functors,* and not functions, with the double dispatcher.

Functors (Chapter 5) are classes that overload the function call operator, `operator()`, thus imitating simple functions in call syntax. Additionally, functors can use member variables for storing and accessing state. Unfortunately, the trampoline trick cannot work with functors, precisely because functors hold state and simple functions do not. (Where would the trampoline hold the state?)

Client code can use `BasicDispatcher` directly, instantiated with the appropriate functor type.

```
struct HatchFunctor
{
   void operator()(Shape&, Shape&)
   {
      ...
   }
};

typedef BasicDispatcher<Shape, Shape, void, HatchFunctor>
   HatchingDispatcher;
```

`HatchFunctor::operator()` cannot be virtual itself, because `BasicDispatcher` needs a functor with value semantics, and value semantics don't mix nicely with runtime polymorphism. However, `HatchFunctor::operator()` can forward a call to a virtual function.

The real disadvantage is that the client loses some automation that the dispatcher could do—namely, taking care of the casts and providing symmetry.

Seems like we're back to square one, but only if you haven't read Chapter 5 on generalized functors. Chapter 5 defines a `Functor` class template that's able to aggregate any kind of functor and pointers to functions, even other `Functor` objects. You can even define specialized `Functor` objects by deriving from the `FunctorImpl` class. We can define a `Functor` to take care of the casts. Once the casts are confined to the library, we can implement symmetry easily.

Let's define a `FunctorDispatcher` that dispatches to any `Functor` objects. This dispatcher will aggregate a `BasicDispatcher` that stores `Functor` objects.

```
template <class BaseLhs, class BaseRhs = BaseLhs, ResultType = void>
class FunctorDispatcher
{
   typedef Functor<ResultType,
         TYPELIST_2(BaseLhs&, BaseRhs&)>
      FunctorType;
   typedef BasicDispatcher<BaseLhs, BaseRhs, ResultType,
         FunctorType>
      BackEndType;
   BackEndType backEnd_;
public:
   ...
};
```

FunctorDispatcher uses a BasicDispatcher instantiation as its back end. BasicDispatcher stores objects of type FunctorType, which are Functors that accept two parameters (BaseLhs and BaseRhs) and return a ResultType.

The FunctorDispatcher::Add member function defines a specialized Functor Impl class by deriving from it. The specialized class (Adapter, shown below) takes care of casting the arguments to the right types; in other words, it *adapts* the argument types from BaseLhs and BaseRhs to SomeLhs and SomeRhs.

```
template <class BaseLhs, class BaseRhs = BaseLhs,
   ResultType = void>
class FunctorDispatcher
{
   ... as above ...
   template <class SomeLhs, class SomeRhs, class Fun>
   void Add(const Fun& fun)
   {
      typedef
         FunctorImpl<ResultType, TYPELIST_2(BaseLhs&, BaseRhs&)>
         FunctorImplType;
      class Adapter : public FunctorImplType
      {
         Fun fun_;
         virtual ResultType operator()(BaseLhs& lhs, BaseRhs& rhs)
         {
            return fun_(
               dynamic_cast<SomeLhs&>(lhs),
               dynamic_cast<SomeRhs&>(rhs));
         }
         virtual FunctorImplType* Clone()const
         { return new Adapter; }
      public:
         Adapter(const Fun& fun) : fun_(fun) {}
      };
      backEnd_.Add<SomeLhs, SomeRhs>(
         FunctorType((FunctorImplType*)new Adapter(fun));
   }
};
```

The Adapter class does exactly what the trampoline function did. Because functors have state, Adapter aggregates a Fun object—something that was impossible with a simple trampoline function. The Clone member function, with obvious semantics, is required by Functor.

FunctorDispatcher::Add has remarkably broad uses. You can use it to register not only pointers to functions, but also just about any functor you want, even generalized functors. The only requirements for the Fun type in Add is that it accept the function-call operator with arguments of types SomeLhs and SomeRhs and that it return a type convertible to ResultType. The following example registers two different functors to a FunctorDispatcher object.

```
typedef FunctorDispatcher<Shape> Dispatcher;
struct HatchRectanglePoly
{
   void operator()(Rectangle& r, Poly& p)
   {
      ...
   }
};
struct HatchEllipseRectangle
{
   void operator()(Ellipse& e, Rectangle& r)
   {
      ...
   }
};
...
Dispatcher disp;
disp.Add<Rectangle, Poly>(HatchRectanglePoly());
disp.Add<Ellipse, Rectangle>(HatchEllipseRectangle());
```

The two functors don't have to be related in any way (like inheriting from a common base). All they have to do is to implement `operator()` for the types that they advertise to handle.

Implementing symmetry with `FunctorDispatcher` is similar to implementing symmetry in `FnDispatcher`. `FunctorDispatcher::Add` defines a new `Reverse Adapter` object that does the casts and reverses the order of calls.

11.9 Converting Arguments: `static_cast` or `dynamic_cast`?

All the previous code has performed casting with the safe `dynamic_cast`. But in the case of `dynamic_cast`, safety comes at a cost in runtime efficiency.

At registration time, you already know that your function or functor will fire for a pair of specific, known types. Through the mechanism it implements, the double dispatcher *knows* the actual types when an entry in the map is found. It seems a waste, then, to have `dynamic_cast` check again for correctness when a simple `static_cast` achieves the same result in much shorter time.

There are, however, two cases in which `static_cast` may fail and the only cast to rely on is `dynamic_cast`. The first occurs when using virtual inheritance. Consider the following class hierarchy:

```
class Shape { ... };
class Rectangle : virtual public Shape { ... };
class RoundedShape : virtual public Shape { ... };
class RoundedRectangle : public Rectangle,
   public RoundedShape { ... };
```

Figure 11.2 displays a graphical representation of the relationships between classes in this hierarchy.

This may not be a very smart class hierarchy, but one thing about designing class libraries is that you never know what your clients might need to do. There are definitely

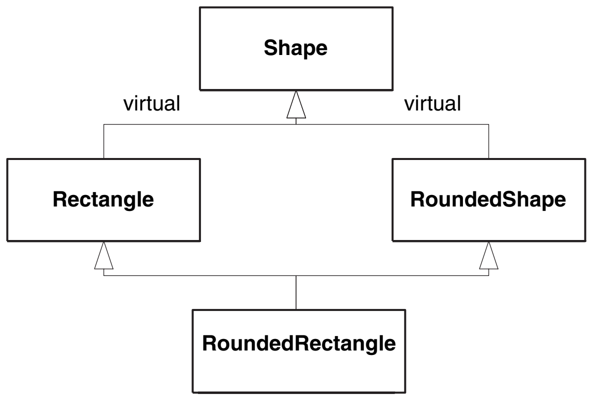

Figure 11.2: A diamond-shaped class hierarchy using virtual inheritance

reasonable situations in which a diamond-shaped class hierarchy is needed, in spite of all its caveats. Consequently, the double dispatchers we defined should work with diamond-shaped class hierarchies.

The dispatchers actually work fine as of now. But if you try to replace the `dynamic_casts` with `static_casts`, you will get compile-time errors whenever you try to cast a `Shape&` to any of `Rectangle&`, `RoundedShape&`, and `RoundedRectangle&`. The reason is that virtual inheritance works very differently from plain inheritance. Virtual inheritance provides a means for several derived classes to share the same base class object. The compiler cannot just lay out a derived object in memory by gluing together a base object with whatever the derived class adds.

In some implementations of multiple inheritance, each derived object stores a pointer to its base object. When you cast from derived to base, the compiler uses that pointer. But the base object does not store pointers to its derived objects. From a pragmatic viewpoint, this all means that after you cast an object of derived type to a virtual base type, there's no compile-time mechanism for getting back to the derived object. You cannot `static_cast` from a virtual base to an object of derived type.

However, `dynamic_cast` uses more advanced means to retrieve the relationships between classes and works nicely even in the presence of virtual bases. In a nutshell, you must use `dynamic_cast` if you have a hierarchy using virtual inheritance.

Second, let's analyze the situation with a similar class hierarchy, but one that doesn't use virtual inheritance—only plain multiple inheritance.

```
class Shape { ... };
class Rectangle : public Shape { ... };
class RoundedShape : public Shape { ... };
class RoundedRectangle : public Rectangle,
   public RoundedShape { ... };
```

Figure 11.3 shows the resulting inheritance graph.

Although the shape of the class hierarchy is the same, the structure of the objects is very different. `RoundedRectangle` now has two distinct subobjects of type `Shape`. This means

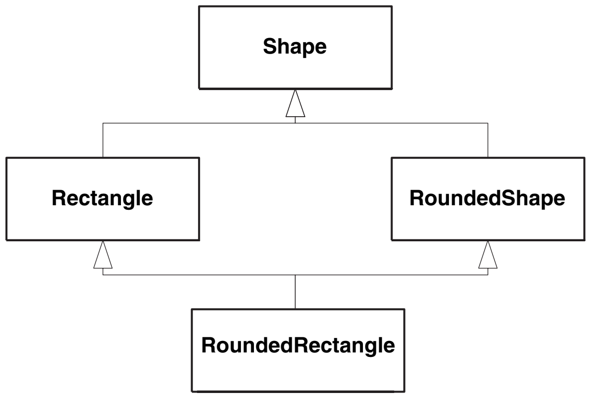

Figure 11.3: A diamond-shaped class hierarchy using nonvirtual inheritance

that converting from RoundedRectangle to Shape is now ambiguous: Which Shape do you mean—that in the RoundedShape or that in the Rectangle? Similarly, you cannot even static cast a Shape& to a RoundedRectangle& because the compiler doesn't know which Shape subobject to consider.

We're facing trouble again. Consider the following code:

```
RoundedRectangle roundRect;
Rectangle& rect = roundRect; // Unambiguous implicit conversion
Shape& shape1 = rect;
RoundedShape& roundShape = roundRect; // Unambiguous implicit
                      // conversion
Shape& shape2 = roundShape;
SomeDispatcher d;
Shape& someOtherShape = ...;
d.Go(shape1, someOtherShape);
d.Go(shape2, someOtherShape);
```

Here, it is essential that the dispatcher use dynamic_cast to convert the Shape& to a RoundedShape&. If you try to register a trampoline for converting a Shape& to a RoundedRectangle&, a compile-time error occurs due to ambiguity.

There is no trouble at all if the dispatcher uses dynamic_cast. A dynamic_cast<RoundedRectangle&> applied to any of the two base Shape subobjects of a RoundedRectangle leads to the correct object. As you can see, nothing beats a dynamic cast. The dynamic_cast operator is designed to reach the right object in a class hierarchy, no matter how intricate its structure is.

The conclusion that consolidates these findings is: You cannot use static_cast with a double dispatcher in a class hierarchy that has multiple occurrences of the same base class, be it through virtual inheritance or not.

This might give you a strong incentive to use dynamic_cast in all dispatchers. However, there are two supplemental considerations.

- Very few class hierarchies in the real world foster a diamond-shaped inheritance graph. Such class hierarchies are very complicated, and their problems tend to outweigh their advantages. That's why most designers avoid them whenever possible.
- `dynamic_cast` is much slower than `static_cast`. Its power comes at a cost. There are many clients who have simple class hierarchies and who require high speed. Committing the double dispatcher to `dynamic_cast` leaves these clients with two options: Reimplement the whole dispatcher from scratch, or rely on some embarrassing surgery into library code.

The solution that Loki adopts is to make casting a *policy*—CastingPolicy. (Refer to Chapter 1 for a description of policies.) Here, the policy is a class template with two parameters, the source and the destination type. The only function the policy exposes is a static function called `Cast`. The following is the `DynamicCaster` policy class.

```
template <class To, class From>
struct DynamicCaster
{
   static To& Cast(From& obj)
   {
      return dynamic_cast<To&>(obj);
   }
};
```

The dispatchers `FnDispatcher` and `FunctorDispatcher` use CastingPolicy according to the guidelines described in Chapter 1. Here is the modified `FunctorDispatcher` class. The changes are shown in bold.

```
template
<
   class BaseLhs,
   class BaseRhs = BaseLhs,
   ResultType = void,
   template <class, class> class CastingPolicy = DynamicCaster
>
class FunctorDispatcher
{
   ...
   template <class SomeLhs, class SomeRhs, class Fun>
   void Add(const Fun& fun)
   {
      class Adapter : public FunctorType::Impl
      {
         Fun fun_;
         virtual ResultType operator()(BaseLhs& lhs,
            BaseRhs& rhs)
         {
            return fun_(
               CastingPolicy<SomeLhs, BaseLhs>::Cast(lhs),
               CastingPolicy<SomeRhs, BaseRhs>::Cast(rhs));
         }
         ... as before ...
      };
```

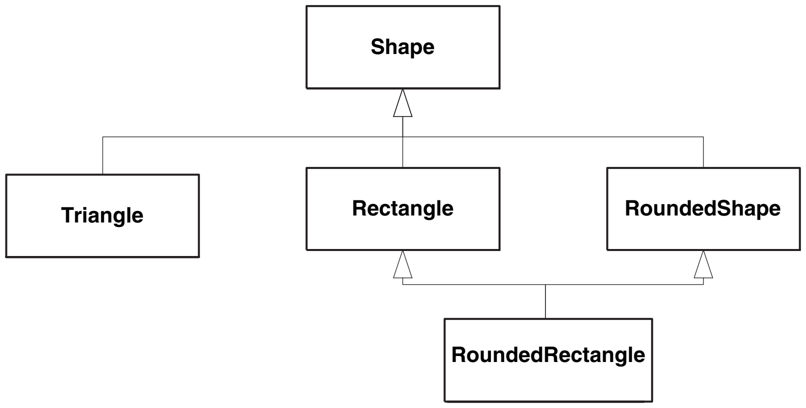

Figure 11.4: A class hierarchy with a diamond-shaped portion

```
         backEnd_.Add<SomeLhs, SomeRhs>(
            FunctorType(new Adapter(fun));
      }
   };
```

Cautiously, the casting policy defaults to DynamicCaster.

Finally, you can do a very interesting thing with casting policies. Consider the hierarchy in Figure 11.4. There are two categories of casts within this hierarchy. Some do not involve a diamond-shaped structure, so static_cast is safe. Namely, static_cast suffices for casting a Shape& to a Triangle&. On the other hand, you cannot static_ cast a Shape& to Rectangle& and any of its derivatives; you must use dynamic_cast.

Suppose you define your own casting policy template for this class hierarchy, namely ShapeCast. You can make it default to dynamic_cast. You can then specialize the policy for the special cases, like so:

```
template <class To, class From>
struct ShapeCaster
{
   static To& Cast(From& obj)
   {
      return dynamic_cast<To&>(obj);
   }
};

template<>
class ShapeCaster<Triangle, Shape>
{
   static Triangle& Cast(Shape& obj)
   {
      return static_cast<Triangle&>(obj);
   }
};
```

You now get the best of both worlds—speedy casts whenever you can, and safe casts whenever you must.

11.10 Constant-Time Multimethods: Raw Speed

Maybe you considered the static dispatcher but found it too coupled, tried the map-based dispatcher but found it too slow. You cannot settle for less: You need absolute speed and absolute scalability, and you're ready to pay for it.

The price to pay in this case is changing your classes. You are willing to allow the double dispatcher to plant some hooks in your classes so that it leverages them later.

This opportunity gives a fresh perspective to implementing a double-dispatch engine. The support for casts remains the same. The means of storing and retrieving handlers must be changed, however—logarithmic time is not constant time.

To find a better dispatching mechanism, let's ask ourselves again: What is double dispatching? You can see it as finding a handler function (or functor) in a two-dimensional space. On one axis are the types of the left-hand operator. On the other axis are the types of the right-hand operator. At the intersection between two types, you find their respective handler function. Figure 11.5 illustrates double dispatch for two class hierarchies—one of `Shapes` and one of `DrawingDevices`. The handlers can be drawing functions that know how to render each concrete `Shape` object on each concrete `DrawingDevice` object.

It doesn't take long to figure out that if you need constant-time searches in this two-dimensional space, you must rely on indexed access in a two-dimensional matrix.

The idea takes off swiftly. Each class must bear a unique integral value, which is the index in the dispatcher matrix. That integral value must be accessible for each class in constant time. A virtual function can help here. When you issue a double-dispatch call, the dispatcher fetches the two indices from the two objects, accesses the handler in the matrix, and launches the handler. Cost: two virtual calls, one matrix indexing operation, and a call through a pointer to a function. The cost is constant.

It seems as if the idea should work quite nicely, but some details of it are not easy to get right. For instance, maintaining indices is very likely to be uncomfortable. For each class, you must assign a unique integral ID and hope that you can detect any duplicates at compile time. The integral IDs must start from zero and have no gaps—otherwise, we would waste matrix storage.

A much better solution is to move index management to the dispatcher itself. Each class stores a static integral variable; initially, its value is −1, meaning "unassigned." A virtual function returns a reference to that static variable, allowing the dispatcher to change it at runtime. As you add new handlers to the matrix, the dispatcher accesses the ID and, if it is −1, assigns the next available slot in the matrix to it.

Here's the gist of this implementation—a simple macro that you must plant in each class of your class hierarchy.

```
#define IMPLEMENT_INDEXABLE_CLASS(SomeClass)
   static int& GetClassIndexStatic()\
   {\
      static int index = -1;\
      return index;\
```

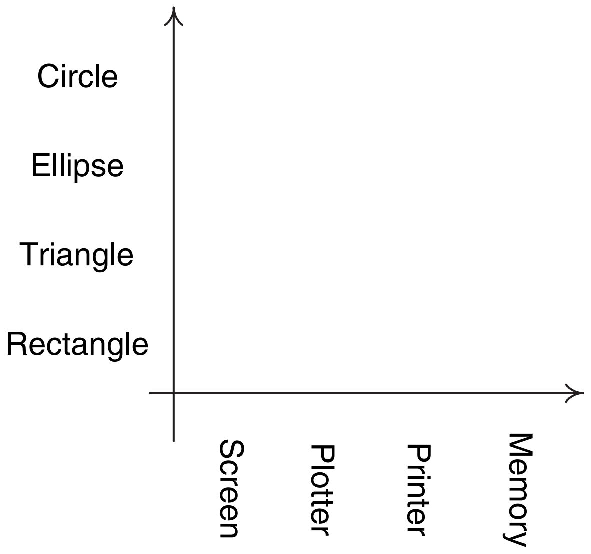

Figure 11.5: Dispatching on `Shapes` and `DrawingDevices`

```
   }\
   virtual int& GetClassIndex()\
   {\
      assert(typeid(*this) == typeid(SomeClass));\
      return GetClassIndexStatic();\
   }
```

You must insert this macro in the `public` portion of each class for which you want to support multiple dispatch.[6]

The `BasicFastDispatcher` class template exposes exactly the same functionality as the previously defined `BasicDispatcher` but uses different storage and retrieval mechanisms.

```
template
<
   class BaseLhs,
   class BaseRhs = BaseLhs,
   typename ResultType = void,
   typename CallbackType = ResultType (*)(BaseLhs&, BaseRhs&)
>
class BasicFastDispatcher
{
   typedef std::vector<CallbackType> Row;
   typedef std::vector<Row> Matrix;
   Matrix callbacks_;
```

[6]Yes, *multiple,* not only double, dispatch. You can easily generalize the index-based solution to support multiple dispatch.

```
      int columns_;
   public:
      BasicFastDispatcher() : columns_(0) {}
      template <class SomeLhs, SomeRhs>
      void Add(CallbackType pFun)
      {
         int& idxLhs = SomeLhs::GetClassIndexStatic();
         if (idxLhs < 0)
         {
            callbacks_.push_back(Row());
            idxLhs = callbacks_.size() - 1;
         }
         else if (callbacks_.size() <= idxLhs)
         {
            callbacks_.resize(idxLhs + 1);
         }
         Row& thisRow = callbacks_[idxLhs];
         int& idxRhs = SomeRhs::GetClassIndexStatic();
         if (idxRhs < 0)
         {
            thisRow.resize(++columns_);
            idxRhs = thisRow.size() - 1;
         }
         else if (thisRow.size() <= idxRhs)
         {
            thisRow.resize(idxRhs + 1);
         }
         thisRow[idxRhs] = pFun;
      }
   };
```

The callback matrix is implemented as a vector of vectors of `MappedType`. The `BasicFastDispatcher::Add` function performs the following sequence of actions:

1. Fetches the ID of each class by calling `GetClassIndexStatic`.
2. Performs initialization and adjustments if one or both indices were not initialized. For uninitialized indices, `Add` expands the matrix to accommodate one extra element.
3. Inserts the callback at the correct position in the matrix.

The `columns_` member variable tallies the number of columns added so far. Strictly speaking, `columns_` is redundant; a search for the maximum row length in the matrix would yield the same result. However, `column_`'s convenience justifies its presence.

The `BasicFastDispatcher::Go` is easy to implement now. The main difference is that `Go` uses the virtual function `GetClassIndex`.

```
   template <...>
   class BasicFastDispatcher
   {
      ... as above ...
      ResultType Go(BaseLhs& lhs, BaseRhs& rhs)
      {
         int& idxLhs = lhs.GetClassIndex();
```

```
        int& idxRhs = rhs.GetClassIndex();
        if (idxLhs < 0 || idxRhs < 0 ||
           idxLhs >= callbacks_.size() ||
           idxRhs >= callbacks_[idxLhs].size() ||
           callbacks_[idxLhs][idxRhs] == 0)
        {
           ... error handling goes here ...
        }
        return callbacks_[idxLhs][idxRhs].callback_(lhs, rhs);
     }
  };
```

Let's recap this section. We defined a matrix-based dispatcher that reaches callback objects in constant time by assigning an integral index to each class. In addition, it performs automatic initialization of its support data (the indices corresponding to the classes). Users of `BasicFastDispatcher` must add a one-macro line, `IMPLEMENT_INDEXABLE_CLASS(`*`YourClass`*`)`, to each class that is to use `BasicFastDispatcher`.

11.11 BasicDispatcher and BasicFastDispatcher as Policies

`BasicFastDispatcher` (matrix based) is preferable to `BasicDispatcher` (map based) when speed is a concern. However, the nice advanced classes `FnDispatcher` and `FunctorDispatcher` are built around `BasicDispatcher`. Should we develop two new classes, `FnFastDispatcher` and `FunctorFastDispatcher`, that use `BasicFastDispatcher` as their back end?

A better idea is to try to adapt `FnDispatcher` and `FunctorDispatcher` to use either `BasicDispatcher` or `BasicFastDispatcher`, depending on a template parameter. That is, make the dispatcher a *policy* for the classes `FnDispatcher` and `FunctorDispatcher`, much as we did with the casting strategy.

The task of morphing the dispatcher into a policy is eased by the fact that `BasicDispatcher` and `BasicFastDispatcher` have the same call interface. This makes replacing one with the other as easy as changing a template argument.

The following is the revised declaration of `FnDispatcher` (`FunctorDispatcher`'s declaration is similar). The changes are shown in bold.

```
template
<
   class BaseLhs,
   class BaseRhs = BaseLhs,
   typename ResultType = void,
   template <class, class>
      class CastingPolicy = DynamicCaster,
   template <class, class, class, class>
      class DispatcherBackend = BasicDispatcher
>
class FnDispatcher; // similarly for FunctorDispatcher
```

Table 11.1 `DispatcherBackend` Policy Requirements

Expression	*Return Type*	*Notes*
`copy`, `assign`, `swap`, `destroy`		Value semantics.
`backEnd.Add<SomeLhs, SomeRhs>(callback)`	`void`	Add a callback to the `backEnd` object for types `SomeLhs` and `SomeRhs`.
`backEnd.Go(BaseLhs&, BaseRhs&)`	`ResultType`	Performs a lookup and a dispatch for the two objects. Throws `std::runtime_error` if a handler is not found.
`backEnd.Remove<SomeLhs, SomeRhs>()`	`bool`	Removes the callback for the types `SomeLhs` and `SomeRhs`. Returns `true` if there was a callback.
`backEnd.HandlerExists <SomeLhs, SomeRhs>()`	`bool`	Returns `true` if a callback is registered for the types `SomeLhs` and `SomeRhs`. No callback is added.

The two classes themselves undergo very few changes.

Let's clarify the DispatcherBackend policy requirements. First of all, obviously, DispatcherBackend must be a template with four parameters. The parameter semantics are, in order:

- Left-hand operand type
- Right-hand operand type
- Return type of the callback
- The callback type

In Table 11.1, `BackendType` represents an instantiation of the dispatcher back-end template, and `backEnd` represents a variable of that type. The table contains functions that we didn't mention yet—don't worry. A complete dispatcher must come with functions that remove callbacks and that do a "passive" lookup without calling the callback. These are trivial to implement; you can see them in Loki's source code, file `MultiMethods.h`.

11.12 Looking Forward

Generalization is right around the corner. We can take our findings regarding double dispatch and apply them to implementing true generic multiple dispatch.

It's actually quite easy. This chapter defines three types of double dispatchers:

- A static dispatcher, driven by two typelists
- A map-based dispatcher, driven by a map keyed by a pair of `std::type_info` objects[7]
- A matrix-based dispatcher, driven by a matrix indexed with unique numeric class IDs

It's easy to generalize these dispatchers as follows. You can generalize the static dispatcher to one driven by a typelist of typelists, instead of two typelists. Yes, you can define a typelist of typelists because any typelist is a type. The following `typedef` defines a typelist of three typelists, possible participants in a triple-dispatch scenario. Remarkably, the resulting typelist is actually easy to read.

```
typedef TYPELIST_3
(
   TYPELIST_3(Shape, Rectangle, Ellipse),
   TYPELIST_3(Screen, Printer, Plotter),
   TYPELIST_3(File, Socket, Memory)
)
ListOfLists;
```

You can generalize the map-based dispatcher to one that is keyed by a vector of `std::type_info` objects (as opposed to a `std::pair`). That vector's size will be the number of objects involved in the multiple-dispatch operation. A possible synopsis of a generalized `BasicDispatcher` is as follows:

```
template
<
   class ListOfTypes,
   typename ResultType,
   typename CallbackType
>
class GeneralBasicDispatcher;
```

The `ListOfTypes` template parameter is a typelist containing the base types involved in the multiple dispatch. For instance, our earlier example of hatching intersections between two shapes would have used a `TYPELIST_2(Shape, Shape)`.

You can generalize the matrix-based dispatcher by using a multidimensional array. You can build a multidimensional array with a recursive class template. The existing scheme of assigning numeric IDs to types works just as it is. This has the nice effect that if you modify a hierarchy once to support double dispatch, you don't have to modify it again to support multiple dispatch.

All these possible extensions need the usual amount of work to get all the details right. A particularly nasty problem related to multiple dispatch and C++ is that there's no uniform way to represent functions with a variable number of arguments.

As of now, Loki implements double dispatch only. The interesting generalizations just suggested are left in the dreaded form of the exercise for . . . you know.

[7]Dressed as `OrderedTypeInfo` to ease comparisons and copying.

11.13 Summary

Multimethods are generalized virtual functions. Whereas the C++ runtime support dispatches virtual functions on a per-class basis, multimethods are dispatched depending on multiple classes at once. This allows you to implement virtual functions for collections of types instead of one type at a time.

By their nature, multimethods are best implemented as a language feature. C++ lacks such a feature, but there are several ways to implement it in libraries.

Multimethods are needed in applications that call algorithms that depend on the type of two or more objects. Typical examples include collisions between polymorphic objects, intersections, and displaying objects on various target devices.

This chapter limits discussion to defining multimethods for two objects. An object that takes care of selecting the appropriate function to call is called a double dispatcher. The types of dispatchers discussed are as follows:

- *The brute-force dispatcher.* This dispatcher relies on static type information (provided in the form of a typelist) and does a linear unrolled search for the correct types. Once the types are found, the dispatcher calls an overloaded member function in a handler object.
- *The map-based dispatcher.* This uses a map keyed by `std::type_info` objects. The mapped value is a callback (either a pointer to a function or a functor). The type discovery algorithm performs a binary search.
- *The constant-time dispatcher.* This is the fastest dispatcher of all, but it requires you to modify the classes on which it acts. The change is to add a macro to each class that you want to use with the constant-time dispatcher. The cost of a dispatch is two virtual calls, a couple of numeric tests, and a matrix element access.

On top of the last two dispatchers, higher-level facilities can be implemented:

- *Automated conversions.* (Not to be confused with automatic conversions.) Because of their uniformity, the dispatchers above require the client to cast the objects from their base types to their derived types. A casting layer can provide a trampoline function that takes care of these conversions.
- *Symmetry.* Some double-dispatch applications are symmetric in nature. They dispatch on the same base type on both sides of the double-dispatch operation, and they don't care about the order of elements. For instance, in a collision detector it doesn't matter whether a spaceship hits a torpedo or a torpedo hits a spaceship—the behavior is the same. Implementing support for symmetry in the library makes client code smaller and less exposed to errors.

The brute-force dispatcher supports these higher-level features directly. This is possible because the brute-force dispatcher has extensive type information available. The other two dispatchers use different methods and add an extra layer to implement automated conversions and symmetry. Double dispatchers for functions implement this extra layer differently (and more efficiently) than double dispatchers for functors.

Table 11.2 compares the three dispatcher types defined in this chapter. As you can see,

Table 11.2: Comparison of Various Implementations of Double Dispatch

	Static Dispatcher **(`StaticDispatcher`)**	***Logarithmic Dispatcher*** **(`BasicDispatcher`)**	***Constant-Time Dispatcher*** **(`BasicFastDispatcher`)**
Speed for few classes	Best	Modest	Good
Speed for many classes	Low	Good	Best
Dependency introduced	Heavy	Low	Low
Alteration of existing classes needed	None	None	Add a macro to each class
Compile-time safety	Best	Good	Good
Runtime safety	Best	Good	Good

none of the presented implementations is ideal. You should choose the solution that best fits your needs for a given situation.

11.14 Double Dispatcher Quick Facts

- Loki defines three basic double dispatchers: `StaticDispatcher`, `BasicDispatcher`, and `BasicFastDispatcher`.
- `StaticDispatcher`'s declaration:

```
template
<
   class Executor,
   class BaseLhs,
   class TypesLhs,
   class BaseRhs = BaseLhs,
   class TypesRhs = TypesLhs,
   typename ResultType = void
>
class StaticDispatcher;
```

where

`BaseLhs` is the base left-hand type.
`TypesLhs` is a typelist containing the set of concrete types involved in the double dispatch on the left-hand side.
`BaseRhs` is the base right-hand type.
`TypesRhs` is a typelist containing the set of concrete types involved in the double dispatch on the right-hand side.

Executor is a class that provides the functions to be invoked after type discovery. Executor must provide an overloaded member function Fire for each combination of types in TypesLhs and TypesRhs.
ResultType is the type returned by the Executor::Fire overloaded functions. The returned value will be forwarded as the result of StaticDispatcher::Go.

- Executor must provide a member function OnError(BaseLhs&, BaseRhs&) for error handling. StaticDispatcher calls Executor::OnError when it encounters an unknown type.
- Example (assume Rectangle and Ellipse inherit Shape, and Printer and Screen inherit OutputDevice):

```
struct Painter
{
   bool Fire(Rectangle&, Printer&);
   bool Fire(Ellipse&, Printer&);
   bool Fire(Rectangle&, Screen&);
   bool Fire(Ellipse&, Screen&);
   bool OnError(Shape&, OutputDevice&);
};

typedef StaticDispatcher
<
   Painter,
   Shape,
   TYPELIST_2(Rectangle, Ellipse),
   OutputDevice,
   TYPELIST_2(Printer&, Screen),
   bool
>
Dispatcher;
```

- StaticDispatcher implements the Go member function, which takes a BaseLhs&, a BaseRhs&, and an Executor&, and executes the dispatch. Example (using the previous definitions):

```
Dispatcher disp;
Shape* pSh = ...;
OutputDevice* pDev = ...;
bool result = disp.Go(*pSh, *pDev);
```

- BasicDispatcher and BasicFastDispatcher implement dynamic dispatchers that allow users to add handler functions at runtime.
- BasicDispatcher finds a handler in logarithmic time. BasicFastDispatcher finds a handler in constant time but requires the user to change the definitions of all dispatched classes.
- Both classes implement the same interface, illustrated here for BasicDispatcher.

```
template
<
   class BaseLhs,
   class BaseRhs = BaseLhs,
   typename ResultType = void,
      typename CallbackType = ResultType (*)(BaseLhs&, BaseRhs&)
   >
   class BasicDispatcher;
```

where:

> `CallbackType` is the type of object that handles the dispatch. `BasicDispatcher` and `BasicFastDispatcher` store and invoke objects of this type.
> All other parameters have the same meaning as for `StaticDispatcher`.

- The two dispatchers implement the functions described in Table 11.1.
- In addition to the three basic dispatchers, Loki also defines two advanced layers: `FnDispatcher` and `FunctorDispatcher`. They use one of `BasicDispatcher` or `BasicFastDispatcher` as a policy.
- `FnDispatcher` and `FunctorDispatcher` have similar declarations, as shown here.

```
   template
   <
      class BaseLhs,
      class BaseRhs = BaseLhs,
      ResultType = void,
      template <class To, class From>
         class CastingPolicy = DynamicCast
      template <class, class, class, class>
         class DispatcherBackend = BasicDispatcher
   >
   class FnDispatcher;
```

where

> `BaseLhs` and `BaseRhs` are the base classes of the two hierarchies involved in the double dispatch.
> `ResultType` is the type returned by the callbacks and the dispatcher.
> `CastingPolicy` is a class template with two parameters. It must implement a static member function `Cast` that accepts a reference to `From` and returns a reference to `To`. The stock implementations `DynamicCaster` and `Static Caster` use `dynamic_cast` and `static_cast`, respectively.
> `DispatcherBackend` is a class template that implements the same interface as `BasicDispatcher` and `BasicFastDispatcher`, described in Table 11.1.

- Both `FnDispatcher` and `FunctorDispatcher` provide an `Add` member function or their primitive handler type. For `FnDispatcher` the primitive handler type is `ResultType(*)(BaseLhs&, BaseRhs&)`. For `FunctorDispatcher`, the primitive handler type is `Functor<ResultType,TYPELIST_2(BaseLhs&, Base Rhs&)>`. Refer to Chapter 5 for a description of `Functor`.
- In addition, `FnDispatcher` provides a template function to register callbacks with the engine:

```
   void Add<SomeLhs, SomeRhs,
      ResultType (*callback)(SomeLhs&, SomeRhs&),
      bool symmetric>();
```

- If you register handlers with the `Add` member function shown in the previous code, you benefit from automated casting and optional symmetry.
- `FunctorDispatcher` provides a template `Add` member function:

```
template <class SomeLhs, class SomeRhs, class F>
void Add(const F& fun);
```

- `F` can be any of the types accepted by the `Functor` object (see Chapter 5), including another `Functor` instantiation. An object of type `F` must accept the function-call operator with arguments of types `BaseLhs&` and `BaseRhs&` and return a type convertible to `ResultType`.
- If no handler is found, all dispatch engines throw an exception of type `std::-runtime_error`.