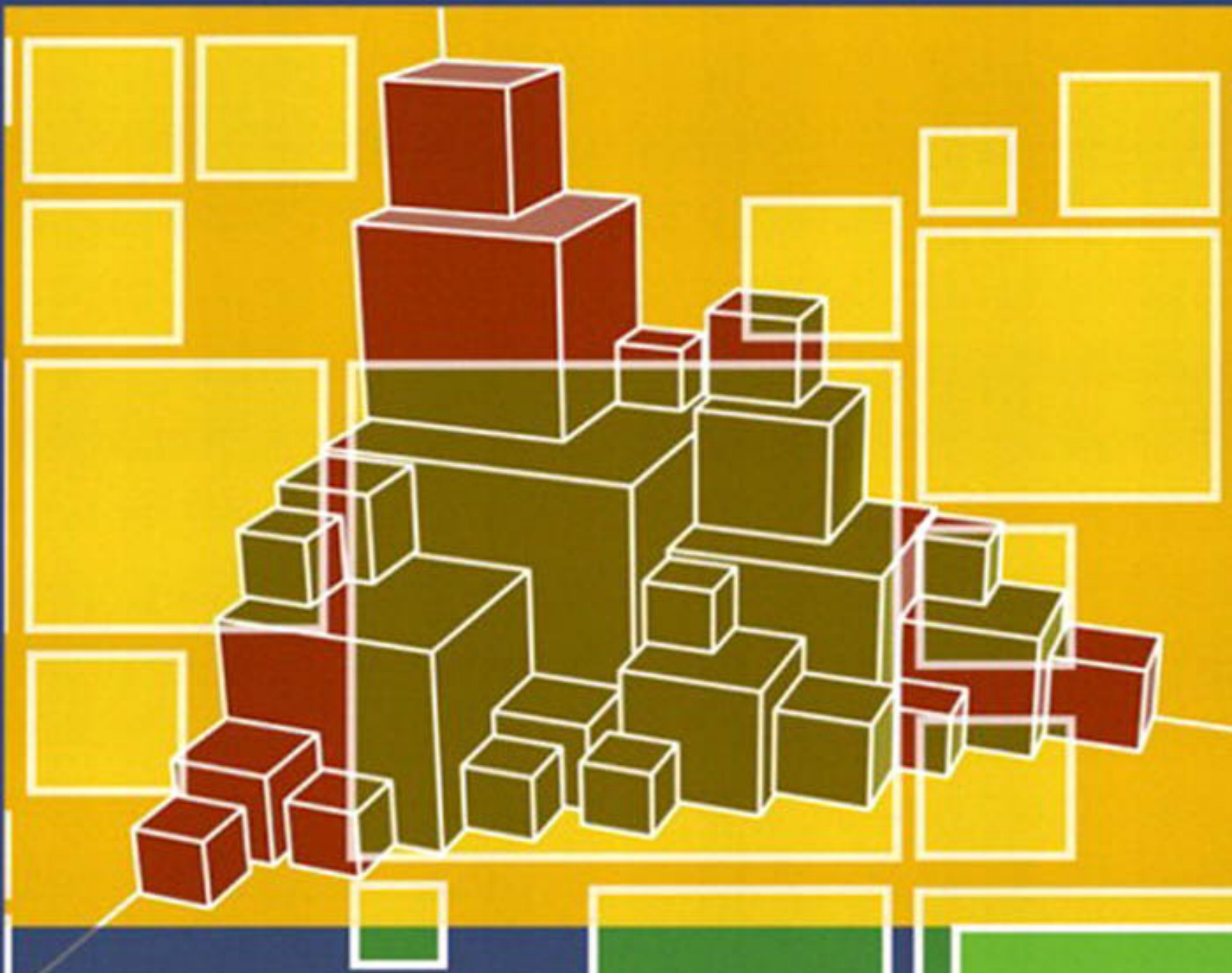
C++ Templates

The Complete Guide



David Vandevoorde
Nicolai M. Josuttis

To Karina
—David

To those who help and love
—Nico

Library of Congress Cataloging-in-Publication Data

Vandevoorde, David.
C++ templates : the complete guide / David Vandevoorde, Nicolai M. Josuttis.
p. cm.
Includes bibliographical references and index.
ISBN 0-201-73484-2 (alk. paper)
1. Microsoft Visual C++. 2. C++ (Computer program language) 3. Standard template library. I. Josuttis, Nicolai M. II. Title.

QA76.73.C153 V37 2003
005.26'8—dc21 2002027933

ISBN 0-201-73484-2

Text printed in the United States on recycled paper at RR Donnelley Crawfordsville in Crawfordsville, Indiana.

12th Printing June 2010

Contents

Appendixes 475

Preface

The idea of templates in C++ is more than ten years old. C++ templates were already documented in 1990 in the "Annotated C++ Reference Manual" or so-called "ARM" (see [*EllisStroustrupARM*]) and they had been described before that in more specialized publications. However, well over a decade later we found a dearth of literature that concentrates on the fundamental concepts and advanced techniques of this fascinating, complex, and powerful C++ feature. We wanted to address this issue and decided to write *the* book about templates (with perhaps a slight lack of humility).

However, we approached the task with different backgrounds and with different intentions. David, an experienced compiler implementer and member of the C++ Standard Committee Core Language Working Group, was interested in an exact and detailed description of all the power (and problems) of templates. Nico, an "ordinary" application programmer and member of the C++ Standard Committee Library Working Group, was interested in understanding all the techniques of templates in a way that he could use and benefit from them. In addition, we both wanted to share this knowledge with you, the reader, and the whole community to help to avoid further misunderstanding, confusion, or apprehension.

As a consequence, you will see both conceptual introductions with day-to-day examples and detailed descriptions of the exact behavior of templates. Starting from the basic principles of templates and working up to the "art of template programming," you will discover (or rediscover) techniques such as static polymorphism, policy classes, metaprogramming, and expression templates. You will also gain a deeper understanding of the C++ standard library, in which almost all code involves templates.

We learned a lot and we had much fun while writing this book. We hope you will have the same experience while reading it. Enjoy!

Chapter 16

Templates and Inheritance

A priori, there might be no reason to think that templates and inheritance interact in interesting ways. If anything, we know from Chapter 9 that deriving from dependent base classes forces us to deal carefully with unqualified names. However, it turns out that some interesting techniques make use of so-called *parameterized inheritance*. In this chapter we describe a few of these techniques.

16.1 Named Template Arguments

Various template techniques sometimes cause a class template to end up with many different template type parameters. However, many of these parameters often have reasonable default values. A natural way to define such a class template may look as follows:

```
template<typename Policy1 = DefaultPolicy1,
         typename Policy2 = DefaultPolicy2,
         typename Policy3 = DefaultPolicy3,
         typename Policy4 = DefaultPolicy4>
class BreadSlicer {
    ...
};
```

Presumably, such a template can often be used with the default template argument values using the syntax `BreadSlicer<>`. However, if a nondefault argument must be specified, all preceding arguments must be specified too (even though they may have the default value).

Clearly, it would be attractive to be able to use a construct akin to `BreadSlicer<Policy3 = Custom>` rather than `BreadSlicer<DefaultPolicy1, DefaultPolicy2, Custom>` as is the case right now. In what follows we develop a technique to enable almost exactly that.[1]

Our technique consists of placing the default type values in a base class and overriding some of them through derivation. Instead of directly specifying the type arguments, we provide them through helper classes. For example, we could write `BreadSlicer<Policy3_is<Custom> >`. Because each template argument can describe any of the policies, the defaults cannot be different. In other words, at a high level every template parameter is equivalent:

```
template <typename PolicySetter1 = DefaultPolicyArgs,
          typename PolicySetter2 = DefaultPolicyArgs,
          typename PolicySetter3 = DefaultPolicyArgs,
          typename PolicySetter4 = DefaultPolicyArgs>
class BreadSlicer {
    typedef PolicySelector<PolicySetter1, PolicySetter2,
                           PolicySetter3, PolicySetter4>
            Policies;
    // use Policies::P1, Policies::P2, ... to refer to the various policies
    ...
};
```

The remaining challenge is to write the `PolicySelector` template. It has to merge the different template arguments into a single type that overrides default typedef members with whichever non-defaults were specified. This merging can be achieved using inheritance:

```
// PolicySelector<A,B,C,D> creates A,B,C,D as base classes
// Discriminator<> allows having even the same base class more than once

template<typename Base, int D>
class Discriminator : public Base {
};

template <typename Setter1, typename Setter2,
          typename Setter3, typename Setter4>
class PolicySelector : public Discriminator<Setter1,1>,
                       public Discriminator<Setter2,2>,
                       public Discriminator<Setter3,3>,
                       public Discriminator<Setter4,4> {
};
```

[1] Note that a similar language extension for function call arguments was proposed (and rejected) earlier in the C++ standardization process (see Section 13.9 on page 216 for details).

Note the use of an intermediate `Discriminator` template. It is needed to allow the various `Setter` types to be identical. (You cannot have multiple direct base classes of the same type. Indirect base classes, on the other hand, can have types that are identical to those of other bases.)

As announced earlier, we're collecting the defaults in a base class:

```
// name default policies as P1, P2, P3, P4
class DefaultPolicies {
  public:
    typedef DefaultPolicy1 P1;
    typedef DefaultPolicy2 P2;
    typedef DefaultPolicy3 P3;
    typedef DefaultPolicy4 P4;
};
```

However, we must be careful to avoid ambiguities if we end up inheriting multiple times from this base class. Therefore, we ensure that the base class is inherited virtually:

```
// class to define a use of the default policy values
// avoids ambiguities if we derive from DefaultPolicies more than once
class DefaultPolicyArgs : virtual public DefaultPolicies {
};
```

Finally, we also need some templates to override the default policy values:

```
template <typename Policy>
class Policy1_is : virtual public DefaultPolicies {
  public:
    typedef Policy P1;  // overriding typedef
};

template <typename Policy>
class Policy2_is : virtual public DefaultPolicies {
  public:
    typedef Policy P2;  // overriding typedef
};

template <typename Policy>
class Policy3_is : virtual public DefaultPolicies {
  public:
    typedef Policy P3;  // overriding typedef
};
```

```
template <typename Policy>
class Policy4_is : virtual public DefaultPolicies {
  public:
    typedef Policy P4;   // overriding typedef
};
```

With all this in place, our desired objective is achieved. Now let's look at what we have by example. Let's instantiate a `BreadSlicer<>` as follows:

```
BreadSlicer<Policy3_is<CustomPolicy> > bc;
```

For this `BreadSlicer<>` the type `Policies` is defined as

```
PolicySelector<Policy3_is<CustomPolicy>,
               DefaultPolicyArgs,
               DefaultPolicyArgs,
               DefaultPolicyArgs>
```

With the help of the `Discriminator<>` class templates this results in a hierarchy, in which all template arguments are base classes (see Figure 16.1). The important point is that these base classes

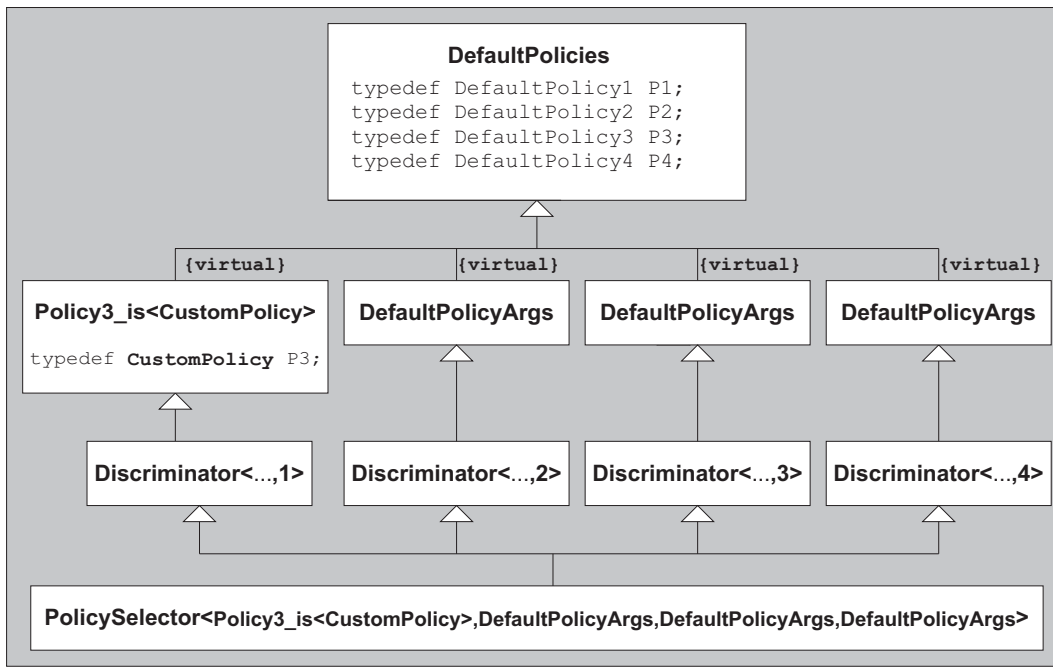


Figure 16.1. Resulting type hierarchy of `BreadSlicer<>::Policies`

all have the same virtual base class `DefaultPolicies`, which defines the default types for `P1`, `P2`, `P3`, and `P4`. However, `P3` is redefined in one of the derived classes—namely, in `Policy3_is<>`. According to the so-called *domination rule* this definition hides the definition of the base class. Thus, this is *not* an ambiguity.[2]

Inside the template `BreadSlicer` you can refer to the four policies by using qualified names such as `Policies::P3`. For example:

```
template <... >
class BreadSlicer {
    ...
  public:
    void print () {
        Policies::P3::doPrint();
    }
    ...
};
```

In `inherit/namedtmpl.cpp` you can find the entire example.

We developed the technique for four template type parameters, but it obviously scales to any reasonable number of such parameters. Note that we never actually instantiate objects of the helper class that contain virtual bases. Hence, the fact that they are virtual bases is not a performance or memory consumption issue.

16.2 The Empty Base Class Optimization (EBCO)

C++ classes are often "empty," which means that their internal representation does not require any bits of memory at run time. This is the case typically for classes that contain only type members, nonvirtual function members, and static data members. Nonstatic data members, virtual functions, and virtual base classes, on the other hand, do require some memory at run time.

Even empty classes, however, have nonzero size. Try the following program if you'd like to verify this:

```
// inherit/empty.cpp

#include <iostream>

class EmptyClass {
};
```

[2] You can find the domination rule in Section 10.2/6 in the C++ Standard (see [*Standard98*]) and a discussion about it in Section 10.1.1 of [*EllisStroustrupARM*].

```
int main()
{
    std::cout << "sizeof(EmptyClass): " << sizeof(EmptyClass)
              << '\n';
}
```

For many platforms, this program will print `1` as size of `EmptyClass`. A few systems impose more strict alignment requirements on class types and may print another small integer (typically, `4`).

16.2.1 Layout Principles

The designers of C++ had various reasons to avoid zero-size classes. For example, an array of zero-size classes would presumably have size zero too, but then the usual properties of pointer arithmetic would not apply anymore. For example, let's assume `ZeroSizedT` is a zero-size type:

```
ZeroSizedT z[10];
...
&z[i] - &z[j]        // compute distance between pointers/addresses
```

Normally, the difference in the previous example is obtained by dividing the number of bytes between the two addresses by the size of the type to which it is pointing, but when that size is zero this is clearly not satisfactory.

However, even though there are no zero-size types in C++, the C++ standard does specify that when an empty class is used as a base class, no space needs to be allocated for it *provided that it does not cause it to be allocated to the same address as another object or subobject of the same type*. Let's look at some examples to clarify what this so-called *empty base class optimization* (or *EBCO*) means in practice. Consider the following program:

```
// inherit/ebco1.cpp

#include <iostream>

class Empty {
    typedef int Int;  // typedef members don't make a class nonempty
};

class EmptyToo : public Empty {
};

class EmptyThree : public EmptyToo {
};
```

```
int main()
{
    std::cout << "sizeof(Empty):      " << sizeof(Empty)
              << '\n';
    std::cout << "sizeof(EmptyToo):   " << sizeof(EmptyToo)
              << '\n';
    std::cout << "sizeof(EmptyThree): " << sizeof(EmptyThree)
              << '\n';
}
```

If your compiler implements the empty base optimization, it will print the same size for every class, but none of these classes has size zero (see Figure 16.2). This means that within class `EmptyToo`, the class `Empty` is not given any space. Note also that an empty class with optimized empty bases (and no other bases) is also empty. This explains why class `EmptyThree` can also have the same size as class `Empty`. If your compiler does not implement the empty base optimization, it will print different sizes (see Figure 16.3).

Figure 16.2. Layout of `EmptyThree` *by a compiler that implements the EBCO*

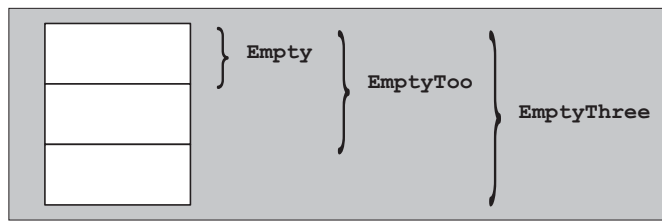


Figure 16.3. Layout of `EmptyThree` *by a compiler that does not implement the EBCO*

Consider an example that runs into a constraint of empty base optimization:

```
// inherit/ebco2.cpp

#include <iostream>

class Empty {
    typedef int Int;  // typedef members don't make a class nonempty
};
```

```
class EmptyToo : public Empty {
};

class NonEmpty : public Empty, public EmptyToo {
};

int main()
{
    std::cout << "sizeof(Empty):    " << sizeof(Empty) << '\n';
    std::cout << "sizeof(EmptyToo): " << sizeof(EmptyToo) << '\n';
    std::cout << "sizeof(NonEmpty): " << sizeof(NonEmpty) << '\n';
}
```

It may come as a surprise that class `NonEmpty` is not an empty class. After all, it does not have any members and neither do its base classes. However, the base classes `Empty` and `EmptyToo` of `NonEmpty` cannot be allocated to the same address because this would cause the base class `Empty` of `EmptyToo` to end up at the same address as the base class `Empty` of class `NonEmpty`. In other words, two subobjects of the same type would end up at the same offset, and this is not permitted by the object layout rules of C++. It may be conceivable to decide that one of the `Empty` base subobjects is placed at offset "0 bytes" and the other at offset "1 byte," but the complete `NonEmpty` object still cannot have a size of one byte because in an array of two `NonEmpty` objects, an `Empty` subobject of the first element cannot end up at the same address as an `Empty` subobject of the second element (see Figure 16.4).

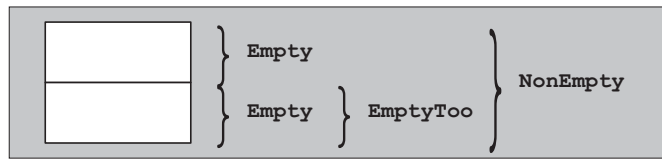


Figure 16.4. Layout of `NonEmpty` *by a compiler that implements the EBCO*

The rationale for the constraint on empty base optimization stems from the fact that it is desirable to be able to compare whether two pointers point to the same object. Because pointers are nearly always internally represented as just addresses, we must ensure that two different addresses (that is, pointer values) correspond to two different objects.

The constraint may not seem very significant. However, in practice, it is often encountered because many classes tend to inherit from a small set of empty classes that define some common typedefs. When two subobjects of such classes are used in the same complete object, the optimization is inhibited.

16.2.2 Members as Base Classes

The empty base class optimization has no equivalent for data members because (among other things) it would create some problems with the representation of pointers to members. As a result, it is sometimes desirable to implement as a (private) base class what would at first sight be thought of as a member variable. However, this is not without its challenges.

The problem is most interesting in the context of templates because template parameters are often substituted with empty class types, but in general one cannot rely on this rule. If nothing is known about a template type parameter, empty base optimization cannot easily be exploited. Indeed, consider the following trivial example:

```
template <typename T1, typename T2>
class MyClass {
  private:
    T1 a;
    T2 b;
    ...
};
```

It is entirely possible that one or both template parameters are substituted by an empty class type. If this is the case, then the representation of `MyClass<T1,T2>` may be suboptimal and may waste a word of memory for every instance of a `MyClass<T1,T2>`.

This can be avoided by making the template arguments base classes instead:

```
template <typename T1, typename T2>
class MyClass : private T1, private T2 {
};
```

However, this straightforward alternative has its own set of problems. It doesn't work when `T1` or `T2` is substituted with a nonclass type or with a union type. It also doesn't work when the two parameters are substituted with the same type (although this can be addressed fairly easily by adding another layer of inheritance; see page 287 or page 449). However, even if we satisfactorily addressed these problems, a very serious problem persists: Adding a base class can fundamentally modify the interface of the given class. For our `MyClass` class, this may not seem very significant because there are very few interface elements to affect, but as we see later in this chapter, inheriting from a template parameter can affect whether a member function is virtual. Clearly, this approach to exploiting EBCO is fraught with all kinds of trouble.

A more practical tool can be devised for the common case when a template parameter is known to be substituted by class types only and when another member of the class template is available. The main idea is to "merge" the potentially empty type parameter with the other member using EBCO. For example, instead of writing

```
template <typename CustomClass>
class Optimizable {
  private:
    CustomClass info;      // might be empty
    void*       storage;
    ...
};
```

a template implementer would use the following:

```
template <typename CustomClass>
class Optimizable {
  private:
    BaseMemberPair<CustomClass, void*> info_and_storage;
    ...
};
```

Even without seeing the implementation of the template `BaseMemberPair`, it is clear that its use makes the implementation of `Optimizable` more verbose. However, various template library implementers have reported that the performance gains (for the clients of their libraries) do justify the added complexity.

The implementation of `BaseMemberPair` can be fairly compact:

```
// inherit/basememberpair.hpp

#ifndef BASE_MEMBER_PAIR_HPP
#define BASE_MEMBER_PAIR_HPP

template <typename Base, typename Member>
class BaseMemberPair : private Base {
  private:
    Member member;
  public:
    // constructor
    BaseMemberPair (Base const & b, Member const & m)
     : Base(b), member(m) {
    }

    // access base class data via first()
    Base const& first() const {
        return (Base const&)*this;
    }
```

```
    Base& first() {
        return (Base&)*this;
    }

    // access member data via second()
    Member const& second() const {
        return this->member;
    }
    Member& second() {
        return this->member;
    }
};

#endif // BASE_MEMBER_PAIR_HPP
```

An implementation needs to use the member functions `first()` and `second()` to access the encapsulated (and possibly storage-optimized) data members.

16.3 The Curiously Recurring Template Pattern (CRTP)

This oddly named pattern refers to a general class of techniques that consists of passing a derived class as a template argument to one of its own base classes. In its simplest form, C++ code for such a pattern looks as follows:

```
template <typename Derived>
class CuriousBase {
    ...
};

class Curious : public CuriousBase<Curious> {
    ...
};
```

Our first outline of CRTP shows a nondependent base class: The class `Curious` is not a template and is therefore immune to some of the name visibility issues of dependent base classes. However, this is not an intrinsic characteristic of CRTP. Indeed, we could just as well have used the following alternative outline:

```
template <typename Derived>
class CuriousBase {
    ...
};

template <typename T>
class CuriousTemplate : public CuriousBase<CuriousTemplate<T> > {
    ...
};
```

From this outline, however, it is not a far stretch to propose yet another alternative formulation, this time involving a template template parameter:

```
template <template<typename> class Derived>
class MoreCuriousBase {
    ...
};

template <typename T>
class MoreCurious : public MoreCuriousBase<MoreCurious> {
    ...
};
```

A simple application of CRTP consists of keeping track of how many objects of a certain class type were created. This is easily achieved by incrementing an integral static data member in every constructor and decrementing it in the destructor. However, having to provide such code in every class is tedious. Instead, we can write the following template:

```
// inherit/objectcounter.hpp

#include <stddef.h>

template <typename CountedType>
class ObjectCounter {
  private:
    static size_t count;      // number of existing objects

  protected:
    // default constructor
    ObjectCounter() {
        ++count;
    }
```

```
    // copy constructor
    ObjectCounter (ObjectCounter<CountedType> const&) {
        ++count;
    }

    // destructor
    ~ObjectCounter() {
        --count;
    }

  public:
    // return number of existing objects:
    static size_t live() {
        return count;
    }
};

// initialize counter with zero
template <typename CountedType>
size_t ObjectCounter<CountedType>::count = 0;
```

If we want to count the number of live (that is, not yet destroyed) objects for a certain class type, it suffices to derive the class from the `ObjectCounter` template. For example, we can define and use a counted string class along the following lines:

```
// inherit/testcounter.cpp

#include "objectcounter.hpp"
#include <iostream>

template <typename CharT>
class MyString : public ObjectCounter<MyString<CharT> > {
  ...
};

int main()
{
    MyString<char> s1, s2;
    MyString<wchar_t> ws;
```

```
    std::cout << "number of MyString<char>:    "
              << MyString<char>::live() << std::endl;
    std::cout << "number of MyString<wchar_t>: "
              << ws.live() << std::endl;
}
```

In general, CRTP is useful to factor out implementations of interfaces that can only be member functions (for example, constructor, destructors, and subscript operators).

16.4 Parameterized Virtuality

C++ allows us to parameterize directly three kinds of entities through templates: types, constants ("nontypes"), and templates. However, indirectly, it also allows us to parameterize other attributes such as the virtuality of a member function. A simple example shows this rather surprising technique:

```
// inherit/virtual.cpp

#include <iostream>

class NotVirtual {
};

class Virtual {
  public:
    virtual void foo() {
    }
};

template <typename VBase>
class Base : private VBase {
  public:
    // the virtuality of foo() depends on its declaration
    // (if any) in the base class VBase
    void foo() {
       std::cout << "Base::foo()" << '\n';
    }
};
```

```
template <typename V>
class Derived : public Base<V> {
  public:
    void foo() {
       std::cout << "Derived::foo()" << '\n';
    }
};

int main()
{
    Base<NotVirtual>* p1 = new Derived<NotVirtual>;
    p1->foo();  // calls Base::foo()

    Base<Virtual>* p2 = new Derived<Virtual>;
    p2->foo();  // calls Derived::foo()
}
```

This technique can provide a tool to design a class template that is usable both to instantiate concrete classes and to extend using inheritance. However, it is rarely sufficient just to sprinkle virtuality on some member functions to obtain a class that makes a good base class for more specialized functionality. This sort of development method requires more fundamental design decisions. It is therefore usually more practical to design two different tools (class or class template hierarchies) rather than trying to integrate them all into one template hierarchy.

16.5 Afternotes

Named template arguments are used to simplify certain class templates in the Boost library. Boost uses metaprogramming to create a type with properties similar to our `PolicySelector` (but without using virtual inheritance). The simpler alternative presented here was developed by one of us (Vandevoorde).

CRTPs have been in use since at least 1991. However, James Coplien was first to describe them formally as a class of so-called *patterns* (see [*CoplienCRTP*]). Since then, many applications of CRTP have been published. The phrase *parameterized inheritance* is sometimes wrongly equated with CRTP. As we have shown, CRTP does not require the derivation to be parameterized at all, and many forms of parameterized inheritance do not conform to CRTP. CRTP is also sometimes confused with the Barton-Nackman trick (see Section 11.7 on page 174) because Barton and Nackman frequently used CRTP in combination with friend name injection (and the latter is an important component of the Barton-Nackman trick). Our `ObjectCounter` example is almost identical to a technique developed by Scott Meyers in [*MeyersCounting*].

Bill Gibbons was the main sponsor behind the introduction of EBCO into the C++ programming language. Nathan Myers made it popular and proposed a template similar to our `BaseMemberPair` to take better advantage of it. The Boost library contains a considerably more sophisticated template, called `compressed_pair`, that resolves some of the problems we reported for the `MyClass` template in this chapter. `boost::compressed_pair` can also be used instead of our `BaseMemberPair`.

Index

Note: Bold page numbers indicate major topics; italic page numbers indicate examples.

C

D

G

H

I

J

K

L

M

N

O

R

S

T

U

V

W

Z