Глава 21

Шаблоны и наследование

Априори нет причины полагать, что шаблоны и наследование взаимодействуют каким-то особым образом. Следует отметить лишь тот факт (см. главу 13, "Имена в шаблонах"), что порождение от зависимых базовых классов требует особой тщательности при использовании неквалифицированных имен. Однако оказывается, что некоторые интересные технологии программирования объединяют эти две возможности языка, включая странно рекурсивный шаблон проектирования (Curiously Recurring Template Pattern — CRTP) и миксины. В этой главе мы бегло рассмотрим эти две технологии.

21.1. Оптимизация пустого базового класса

Классы C++ часто бывают "пустыми", т.е. их внутреннее представление не требует выделения памяти во время работы программы. Это типичное поведение классов, которые содержат только члены-типы, невиртуальные функции-члены и статические члены-данные. Нестатические члены-данные, виртуальные функции и виртуальные базовые классы требуют при работе программы выделения памяти.

Однако даже пустые классы имеют ненулевой размер. Если вы хотите это проверить, попробуйте запустить приведенную ниже программу.

inherit/empty.cpp

```
#include <iostream>

class EmptyClass
{
};

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

На множестве платформ эта программа выведет `1` в качестве размера класса `EmptyClass`. Некоторые системы продемонстрируют более строгие требования к выравниванию и выведут иное небольшое значение (обычно `4`).

21.1.1. Принципы размещения

Проектировщики C++ имели множество причин избегать классов с нулевым размером. Например, массив классов, имеющих нулевые размеры, также имел бы нулевой размер, но при этом арифметика указателей оказалась бы неприменима. Пусть, например, `ZeroSizedT` — тип с нулевым размером:

```
ZeroSizedT z[10];
...
&z[i] - &z[j] // Вычисление расстояния между указателями/адресами
```

Обычно разность из предыдущего примера получается путем деления числа байтов между двумя адресами на размер объекта указываемого типа. Однако, если этот размер нулевой, понятно, что такая операция не приведет к корректному результату.

Тем не менее, даже при том, что в C++ нет типов с нулевым размером, стандарт C++ устанавливает, что, когда пустой класс используется в качестве базового, память для него не выделяется *при условии, что это не приводит к размещению объекта по адресу, где уже расположен другой объект или подобъект того же самого типа*. Рассмотрим несколько примеров, чтобы разъяснить, что означает на практике так называемая *оптимизация пустого базового класса* (empty base class optimization — EBCO). Рассмотрим приведенную ниже программу:

inherit/ebco1.cpp

```
#include <iostream>

class Empty
{
    using Int = int; // Псевдоним типа не делает класс непустым
};

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';
}
```

Если ваш компилятор реализует оптимизацию пустого базового класса, то он выведет один и тот же размер для каждого класса (но ни один из этих классов не будет иметь нулевой размер (рис. 21.1). Это означает, что внутри класса `EmptyToo` классу `Empty` не выделяется никакое пространство. Обратите внимание и на то, что пустой класс с оптимизированными пустыми базовыми классами (при отсутствии непустых базовых классов) также пуст. Это объясняет, почему класс `EmptyThree` может иметь тот же размер, что и класс `Empty`. Если же ваш компилятор не выполняет оптимизацию пустого базового класса, выведенные размеры будут разными (рис. 21.2).

Рис. 21.1. Размещение `EmptyThree` компилятором, который реализует EBCO

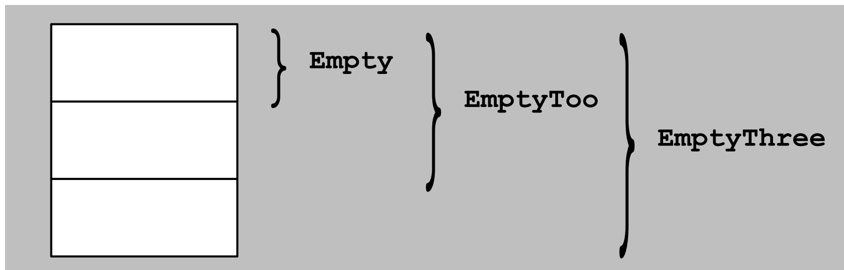

Рис. 21.2. Размещение `EmptyThree` компилятором, который не реализует EBCO

Рассмотрим пример, в котором оптимизация пустого базового класса запрещена:

inherit/ebco2.cpp

```
#include <iostream>

class Empty
{
    using Int = int; // Псевдоним типа не делает класс непустым
};

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';
}
```

Может показаться неожиданным, что класс `NonEmpty` не пустой. Ведь ни он, ни его базовые классы не содержат никаких членов. Но дело в том, что базовые классы `Empty` и `EmptyToo` класса `NonEmpty` не могут быть размещены по одному и тому же адресу, поскольку это привело бы к размещению объекта базового класса `Empty`, принадлежащего классу `EmptyToo`, по тому же адресу, что и объекта базового класса `Empty`, принадлежащего классу `NonEmpty`. Иными словами, два подобъекта одного и того же типа находились бы в одном месте, а это

не разрешено правилами размещения объектов языка C++. Можно решить, что один из базовых подобъектов Empty помещен со смещением 0 байт, а другой — со смещением 1 байт, но полный объект NonEmpty все равно не может иметь размер в один байт, так как в массиве из двух объектов NonEmpty подобъект Empty первого элемента не может находиться по тому же адресу, что и подобъект Empty второго элемента (рис. 21.3):

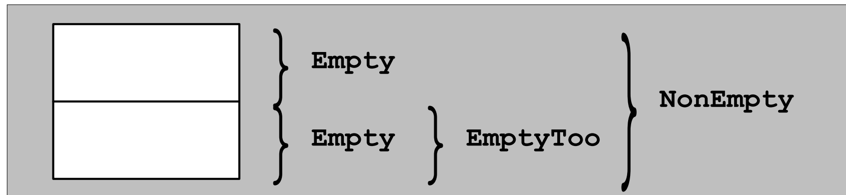

Рис. 21.3. Размещение объекта NonEmpty компилятором, реализующим EBCO

Ограничение на оптимизацию пустого базового класса можно объяснить необходимостью проверки, не указывают ли два указателя на один и тот же объект. Поскольку указатели почти всегда внутренне представлены как обычные адреса, необходимо гарантировать, что два различных адреса соответствуют двум различным объектам.

Это ограничение может показаться не очень существенным, однако с ним часто приходится сталкиваться на практике, поскольку многие классы наследуются из небольшого набора пустых классов, определяющих некоторое общее множество синонимов имен типов. Когда два подобъекта таких классов оказываются в одном и том же полном объекте, оптимизация запрещена.

Даже при этом ограничении EBCO является важной оптимизацией для библиотек шаблонов, потому что ряд методов основываются на введении базовых классов просто с целью введения нового псевдонима типа или предоставления дополнительной функциональности без добавления новых данных. В этой главе будет описано несколько таких методов.

21.1.2. Члены как базовые классы

Оптимизация пустого базового класса не имеет эквивалента для членов-данных, поскольку (помимо прочего) это создало бы ряд проблем с представлением указателей на члены. Поэтому иногда то, что реализовано как данные-члены, желательно реализовать в виде (закрытого) базового класса, который на первый взгляд выглядит как переменная-член. Однако и здесь не обходится без проблем.

Наиболее интересна эта задача в контексте шаблонов, поскольку параметры шаблона часто заменяются типами пустых классов (хотя, конечно, в общем случае полагаться на это нельзя). Если о типовом параметре шаблона ничего не известно, оптимизацию пустого базового класса осуществить не так-то легко. Рассмотрим тривиальный пример:

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

Вполне возможно, что один или оба параметра шаблона заменяются типом пустого класса. В этом случае представление MyClass<T1,T2> может оказаться не оптимальным, что приведет к напрасной трате одного слова памяти для каждого экземпляра MyClass<T1,T2>.

Этого можно избежать, сделав аргументы шаблона базовыми классами:

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

Однако этот простой вариант имеет свои недостатки.

- Он не работает, если T1 или T2 заменяются типом, не являющимся классом или типом объединения.
- Он также не работает, если два параметра заменяются одним и тем же типом (хотя можно легко решить эту проблему, добавив лишний уровень наследования, как было показано ранее в главе).
- Класс может быть объявлен как final, и в этом случае попытки его наследования будут приводить к ошибкам.

Но даже после решения этих проблем адресации останется еще одна очень серьезная неприятность: добавление базового класса может существенно изменить интерфейс данного класса. Для нашего класса MyClass эта проблема может показаться не очень значительной, поскольку здесь совсем немного элементов интерфейса, на которые может воздействовать добавление базового класса, но, как будет показано далее в главе, наличие виртуальных функций-членов меняет картину. Понятно, что рассматриваемый подход к EBCO чреват всеми описанными видами проблем.

Более практичное решение может быть разработано для распространенного случая, когда известно, что параметр шаблона заменяется только типами классов и когда доступен другой член шаблона класса. Основная идея состоит в том, чтобы “слить” потенциально пустой параметр типа с другим членом с использованием EBCO. Например, вместо записи

```
template<typename CustomClass>
class Optimizable
{
  private:
    CustomClass info; // Может быть пустым
    void* storage;
    ...
};
```

можно написать:

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

Даже беглого взгляда достаточно, чтобы понять, что использование шаблона `BaseMemberPair` делает реализацию `Optimizable` более многословной. Однако некоторые разработчики библиотек шаблонов отмечают, что повышение производительности (для клиентов их библиотек) стоит этой дополнительной сложности. Мы рассмотрим эту идиому позже при изучении кортежей в разделе 25.1.1.

Реализация `BaseMemberPair` может быть достаточно компактной:

inherit/basememberpair.hpp

```
#ifndef BASE_MEMBER_PAIR_HPP
#define BASE_MEMBER_PAIR_HPP
template<typename Base, typename Member>
class BaseMemberPair : private Base
{
  private:
    Member mem;
  public:
    // Конструктор
    BaseMemberPair(Base const& b, Member const& m)
        : Base(b), mem(m)
    {
    }
    // Доступ к данным базового класса через base()
    Base const& base() const
    {
        return static_cast<Base const&>(*this);
    }
    Base& base()
    {
        return static_cast<Base&>(*this);
    }
    // Доступ к членам-данным через member()
    Member const& member() const
    {
        return this->mem;
    }
    Member& member()
    {
        return this->mem;
    }
};
#endif // BASE_MEMBER_PAIR_HPP
```

Для доступа к инкапсулированным (и, возможно, оптимизированным с точки зрения расхода памяти) элементам данных реализация должна использовать функции-члены `base()` и `member()`.

21.2. Странно рекурсивный шаблон проектирования

Еще одна идиома, которую стоит рассмотреть — *Странно рекурсивный шаблон проектирования* (Curiously Recurring Template Pattern — CRTP). Это странное название обозначает общий класс методов, которые состоят в передаче класса-наследника в качестве аргумента шаблона одному из собственных базовых классов. В самой простой форме код C++ такой модели выглядит, как показано ниже.

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

Наш первый набросок CRTP имеет независимый базовый класс: `Curious` не является шаблоном и, следовательно, защищен от проблем видимости имен зависимых базовых классов. Однако это не главная характеристика CRTP. Точно так же можно было использовать альтернативную схему:

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

Передавая производный класс базовому в качестве параметра, базовый класс может настроить собственное поведение для данного производного класса так, что не возникнет необходимости в использовании виртуальных функций. Это делает CRTP полезным методом для исключения реализаций, которые могут быть только функциями-членами (например, конструкторы, деструкторы или операторы индексов) или зависят от производного класса.

Простейшее применение CRTP — отслеживание количества созданных объектов некоторого типа класса. Этого легко достичь посредством увеличения целого статического члена-данного в каждом конструкторе и его уменьшения в деструкторе. Однако необходимость обеспечить соответствующий код в каждом классе весьма утомительна, а реализация этой функциональности посредством одного (не-CRTP) базового класса будет спутываться со счетчиками объектов

для различных производных классов. Но вместо этого можно просто написать следующий шаблон:

inherit/objectcounter.hpp

```
#include <cstddef>

template<typename CountedType>
class ObjectCounter
{
  private:
    inline static                // Количество
        std::size_t count = 0;  // существующих объектов
  protected:
    // Конструктор по умолчанию
    ObjectCounter()
    {
        ++count;
    }
    // Копирующий конструктор
    ObjectCounter(ObjectCounter<CountedType> const&)
    {
        ++count;
    }
    // Перемещающий конструктор
    ObjectCounter(ObjectCounter<CountedType>&&)
    {
        ++count;
    }
    // Деструктор
    ~ObjectCounter()
    {
        --count;
    }
  public:
    // Возврат количества имеющихся объектов:
    static std::size_t live()
    {
        return count;
    }
};
```

Обратите внимание на использование `inline`, чтобы иметь возможность определить и инициализировать член `count` внутри структуры класса. До C++17 мы должны были определять его вне шаблона класса:

```
template<typename CountedType>
class ObjectCounter
{
  private:
    static std::size_t count; // Количество существующих объектов
    ...
};

// Инициализация счетчика нулем:
template<typename CountedType>
std::size_t ObjectCounter<CountedType>::count = 0;
```

Если требуется подсчитать количество активных (не уничтоженных) объектов некоторого типа класса, достаточно породить класс из шаблона `ObjectCounter`. Например, можно определить и использовать класс строк с подсчетом объектов.

inherit/countertest.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 << "Количество MyString<char>: "
              << MyString<char>::live() << '\n';
    std::cout << "Количество MyString<wchar_t>: "
              << ws.live() << '\n';
}
```

21.2.1. Метод Бартона–Нэкмана

В 1994 году Джон Бартон (John J. Barton) и Ли Нэкман (Lee R. Nackman) представили метод применения шаблонов, названный ими *ограниченным расширением шаблонов* (restricted template expansion [7]). Частично причиной разработки этого метода послужил тот факт, что в то время шаблоны функций нельзя было перегружать[1], а пространства имен в большинстве компиляторов были недоступны.

Чтобы проиллюстрировать указанный метод, предположим, что у нас есть шаблон класса `Array`, в котором требуется определить оператор равенства ==. Одна из возможностей — объявить этот оператор членом класса. Однако недостаток этого подхода состоит в том, что первый аргумент (связанный с указателем `this`) подчиняется правилам преобразования типов, отличным от тех, которые применимы ко второму аргументу. Поскольку удобнее, чтобы оператор == был симметричным относительно своих аргументов, лучше объявить его как функцию в области видимости пространства имен. Общая схема такого подхода к реализации оператора == может выглядеть следующим образом:

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

[1] Возможно, вам стоит прочитать раздел 16.2, чтобы понять, как в современном C++ работает перегрузка шаблонов функций.

```
template<typename T>
bool operator== (Array<T> const& a, Array<T> const& b)
{
    ...
}
```

Однако если перегрузка шаблонов функций не допускается, возникает проблема: в этом пространстве имен другие шаблоны операторов == объявлять нельзя, хотя они могут понадобиться для иных шаблонов классов. Бартону и Нэкману удалось решить эту проблему путем определения в классе оператора равенства в виде обычной функции-друга.

```
template<typename T>
class Array
{
    static bool areEqual(Array<T> const& a, Array<T> const& b);
  public:
    ...
    friend bool operator== (Array<T> const& a, Array<T> const& b)
    {
        return areEqual(a, b);
    }
};
```

Предположим, что эта версия шаблона `Array` инстанцируется для типа `float`. Тогда в процессе инстанцирования объявляется функция-друг, с помощью которой реализован оператор равенства. Заметим, что сама по себе эта функция не является результатом инстанцирования шаблона функции. Это обычная функция (а не шаблон), *введенная* в глобальную область видимости как побочный эффект процесса инстанцирования. Поскольку это нешаблонная функция, ее можно перегружать с другими объявлениями оператора ==, причем это можно было делать еще до того, как в языке появились шаблоны функций. Бартон и Нэкман дали этому методу название *ограниченного расширения шаблонов*, поскольку в нем не используется шаблон `operator==(T,T)`, применимый для всех типов `T` (другими словами, *не*ограниченное расширение).

Поскольку

```
operator== (Array<T> const&, Array<T> const&)
```

определен в теле класса, он автоматически является встраиваемой (`inline`) функцией, поэтому мы решили делегировать его реализацию статической функции-члену `areEqual`, которая не обязательно должна быть встроенной.

Поиск имен для определений дружественных функций с 1994 года изменился, поэтому метод Бартона–Нэкмана в стандартном C++ не так полезен. Во время его изобретения объявления друзей были видимы в охватывающей области видимости шаблона класса, когда этот шаблон инстанцировался с помощью процесса, который называется *внесение имени друга* (friend name injection). Нынешний стандарт C++ вместо этого находит объявления дружественных функций с помощью поиска, зависящего от аргумента (см. детали в разделе 13.2.2). Это означает, что по крайней мере один из аргументов вызова функции уже должен иметь класс, содержащий дружественную функцию, в качестве связанного. Дружественную

функцию не удастся найти, если аргументы имеют тип несвязанного класса, который может быть преобразован в класс, содержащий друга. Например:

inherit/wrapper.cpp

```
class S
{
};
template<typename T>
class Wrapper
{
  private:
    T object;
  public:
    Wrapper(T obj) // Неявное преобразование T в Wrapper<T>
    : object(obj){}
    friend void foo(Wrapper<T> const&)
    {}
};

int main()
{
    S s;
    Wrapper<S> w(s);
    foo(w); // OK: Wrapper<S> — класс, связанный с w
    foo(s); // ERROR: Wrapper<S> не связан с s
}
```

В данном примере вызов функции `foo(w)` корректен, поскольку функция `foo()` является другом, объявленным в классе `Wrapper<S>`, с которым связана переменная `w`[2]. Однако при вызове `foo(s)` объявление функции-друга `foo(Wrapper<S> const&)` невидимо, поскольку класс `Wrapper<S>`, в котором определена функция `foo()`, не связан с аргументом `s` типа `S`. Поэтому, несмотря на допустимость неявного преобразования типа `S` в тип `Wrapper<S>` (с помощью конструктора класса `Wrapper<S>`), такое преобразование не рассматривается, поскольку функция-кандидат `foo()` не найдена в первую очередь. Во времена изобретения Бартоном и Нэкманом своего метода введение имени друга делало друга `foo()` видимым, а вызов `foo(s)` — успешным.

В современном C++ единственное преимущество при определении дружественной функции в шаблоне класса перед простым определением шаблона обычной функции является сугубо синтаксическим: определения дружественных функций имеют доступ к закрытым и защищенным членам включающего их класса, и не нужно повторять все параметры шаблона охватывающего шаблона класса. Однако определения дружественных функций могут быть полезными при сочетании их со странно рекурсивным шаблоном проектирования, как показано в реализации операторов, описанной в следующем разделе.

[2] Заметим, что эта переменная также связана с классом `S`, поскольку этот класс представляет собой аргумент шаблона типа переменной `w`. Конкретные правила ADL рассматриваются в разделе 13.2.1.

21.2.2. Реализации операторов

При реализации класса, который предоставляет перегруженные операторы, обычно предоставляются перегрузки для целого ряда различных (но взаимосвязанных) операторов. Например, класс, который реализует оператор равенства (==), скорее всего, реализует также оператор неравенства (!=), а класс, реализующий оператор меньше (<), скорее всего, реализует и другие реляционные операторы (>, <=, >=). Во многих случаях фактически интересно только определение одного из этих операторов, в то время как другие просто могут быть определены в терминах этого одного оператора. Например, оператор неравенства для класса X может быть определен с помощью оператора равенства:

```
bool operator!= (X const& x1, X const& x2)
{
    return !(x1 == x2);
}
```

Учитывая большое количество типов с аналогичными определениями оператора !=, возникает соблазн обобщить его в шаблоне:

```
template<typename T>
bool operator!= (T const& x1, T const& x2)
{
    return !(x1 == x2);
}
```

Стандартная библиотека языка C++ и в самом деле содержит такие определения в заголовочном файле <utility>. Однако эти определения (для !=, >, <= и >=) были отнесены к пространству имен std::rel_ops во время стандартизации, когда было установлено, что, будучи доступны в пространстве имен std, они вызывали проблемы. Когда эти определения видимы, получается, что *любой* тип имеет оператор != (который может не инстанцироваться), и что этот оператор всегда будет иметь точное соответствие для обоих своих аргументов. В то время как первая проблема может быть решена с использованием методов SFINAE (см. раздел 19.4), так что это определение оператора != будет инстанцироваться только для типов с подходящим оператором ==, вторая проблема остается нерешенной: общее определение оператора !=, показанное выше, будет предпочтительнее пользовательского определения, которое требует, например, преобразовать производный класс в базовый, что может оказаться неприятным сюрпризом.

Альтернативная формулировка этих шаблонов операторов на основе CRTP позволяет классам прибегать к определениям общих операторов, предоставляя преимущества повышения повторного использования кода без побочных эффектов наличия слишком общего оператора:

inherit/equalitycomparable.cpp

```
template<typename Derived>
class EqualityComparable
{
  public:
```

```
    friend bool operator!= (Derived const& x1, Derived const& x2)
    {
        return !(x1 == x2);
    }
};

class X : public EqualityComparable<X>
{
  public:
    friend bool operator== (X const& x1, X const& x2)
    {
        // Реализация логики сравнения двух объектов типа X
    }
};

int main()
{
    X x1, x2;

    if (x1 != x2) { }
}
```

Здесь мы объединили CRTP с методом Бартона–Нэкмана. Шаблон `EqualityComparable<>` использует CRTP для предоставления оператора != для производного класса на основе определения оператора == производного класса. Это определение фактически предоставлено через определение дружественной функции (метод Бартона–Нэкмана), которая обеспечивает одинаковое поведение обоих параметров оператора != при преобразовании типов.

Применение CRTP может быть полезным при переносе поведения в базовый класс при сохранении идентичности возможного производного класса. Наряду с методом Бартона–Нэкмана идиома CRTP может обеспечить общие определения для ряда операторов на основе некоторых канонических операторов. Эти свойства сделали идиому CRTP с методом Бартона–Нэкмана любимой технологией авторов шаблонных библиотек C++.

21.2.3. Фасады

Использование идиомы CRTP и метода Бартона–Нэкмана для определения некоторых операторов представляет собой удобное сокращение. Мы можем развить эту идею дальше, так, чтобы базовый класс CRTP определял большую часть или весь открытый интерфейс класса в терминах гораздо меньшего (но легче реализуемого) интерфейса, предоставляемого производным классом CRTP. Этот проектный шаблон, именуемый *фасадом*, особенно полезен при определении новых типов, которые должны отвечать требованиям некоторого существующего интерфейса — числовых типов, итераторов, контейнеров и так далее.

Чтобы проиллюстрировать проектный шаблон фасада, мы реализуем фасад для итераторов, который существенно упрощает процесс написания итераторов, соответствующих требованиям стандартной библиотеки. Требуемый интерфейс для типа итератора (особенно *итератора с произвольным доступом*) довольно велик. Приведенная ниже схема для шаблона класса `IteratorFacade` демонстрирует требования к интерфейсу итератора.

inherit/iteratorfacadeskel.hpp

```
template<typename Derived, typename Value, typename Category,
         typename Reference = Value&,
         typename Distance = std::ptrdiff_t>
class IteratorFacade
{
  public:
    using value_type = typename std::remove_const<Value>::type;
    using reference = Reference;
    using pointer = Value*;
    using difference_type = Distance;
    using iterator_category = Category;

    // Интерфейс входного итератора:
    reference operator *() const
    {  ...  }
    pointer operator ->() const
    {  ...  }
    Derived& operator ++()
    {  ...  }
    Derived operator ++(int)
    {  ...  }
    friend bool operator== (IteratorFacade const& lhs,
                            IteratorFacade const& rhs)
    {  ...  }
    ...

    // Интерфейс двунаправленного итератора:
    Derived& operator --()
    {  ...  }
    Derived operator --(int)
    {  ...  }

    // Интерфейс итератора с произвольным доступом:
    reference operator [](difference_type n) const
    {  ...  }
    Derived& operator +=(difference_type n)
    {  ...  }
    ...
    friend difference_type operator -(IteratorFacade const& lhs,
                                      IteratorFacade const& rhs)
    {  ...  }
    friend bool operator <(IteratorFacade const& lhs,
                           IteratorFacade const& rhs)
    {  ...  }
    ...
};
```

Мы полностью опустили некоторые объявления для краткости, но даже реализация всех перечисленных функций для каждого нового итератора является довольно трудоемкой работой. К счастью, этот интерфейс можно свести к нескольким основным операциям.

- Для всех итераторов:
  - `dereference()`: доступ к значению, на которое указывает итератор (обычно используется операторами `*` и `->`);
  - `increment()`: перемещение итератора так, чтобы он указывал на следующий элемент последовательности;
  - `equals()`: определение, указывают ли два итератора на один и тот же элемент последовательности.
- Для двунаправленных итераторов:
  - `decrement()`: перемещение итератора так, чтобы он указывал на предыдущий элемент в последовательности.
- Для итераторов с произвольным доступом:
  - `advance()`: перемещение итератора на *n* шагов вперед (или назад);
  - `measureDistance()`: определение количества шагов для перехода от одного итератора к другому в последовательности.

Фасад призван адаптировать тип, который реализует только указанные основные операции, для предоставления полного интерфейса итератора. Реализация `IteratorFacade` в основном состоит из отображения синтаксиса итератора на этот минимальный интерфейс. В следующих примерах мы используем функции-члены `asDerived()` для доступа к производному классу CRTP:

```
Derived& asDerived()
{
    return *static_cast<Derived*>(this);
}

Derived const& asDerived() const
{
    return *static_cast<Derived const*>(this);
}
```

При наличии такого определения реализация большей части фасада является простой задачей[3]. Мы проиллюстрируем только определения для некоторых требований входных итераторов; прочие определения создаются аналогично.

```
reference operator*() const
{
    return asDerived().dereference();
}

Derived& operator++()
{
    asDerived().increment();
    return asDerived();
}
```

[3] Для упрощения представления мы игнорируем наличие прокси-итераторов, операция разыменования которых не возвращает истинную ссылку. Полная реализация фасада итератора, как в [15], должна корректировать возвращаемые типы `operator->` и `operator[]` для учета наличия прокси.

```
Derived operator++(int)
{
    Derived result(asDerived());
    asDerived().increment();
    return result;
}

friend bool operator== (IteratorFacade const& lhs,
                        IteratorFacade const& rhs)
{
    return lhs.asDerived().equals(rhs.asDerived());
}
```

Определение итератора связанного списка

При наличии нашего определения `IteratorFacade` мы можем легко определить итератор для простого класса связанного списка. Представим, например, что мы определили узел в связанном списке следующим образом:

inherit/listnode.hpp

```
template<typename T>
class ListNode
{
  public:
    T value;
    ListNode<T>* next = nullptr;
    ~ListNode()
    {
        delete next;
    }
};
```

Используя `IteratorFacade`, можно легко определить итератор, работающий с таким списком:

inherit/listnodeiterator0.hpp

```
template<typename T>
class ListNodeIterator
    : public IteratorFacade<ListNodeIterator<T>, T,
                            std::forward_iterator_tag>
{
    ListNode<T>* current = nullptr;
  public:
    T& dereference() const
    {
        return current->value;
    }
    void increment()
    {
        current = current->next;
    }
```

```
    bool equals(ListNodeIterator const& other) const
     {
         return current == other.current;
     }
     ListNodeIterator(ListNode<T>* current = nullptr) : current(
             current) { }
};
```

ListNodeIterator предоставляет все корректные операторы и вложенные типы, необходимые для работы в качестве итератора однонаправленного итератора, и требует очень мало кода для реализации. Как мы увидим позже, определение более сложных итераторов (например, итераторов произвольного доступа) требует лишь небольшого количества дополнительной работы.

Сокрытие интерфейса

Один из недостатков нашей реализации ListNodeIterator заключается в том, что мы обязаны предоставить в качестве открытого интерфейса операции dereference(), advance() и equals(). Чтобы исключить это требование, можно переделать IteratorFacade так, чтобы он выполнял все свои операции над производным классом CRTP через отдельный класс *доступа*, который мы назовем IteratorFacadeAccess:

inherit/iteratorfacadeaccessskel.hpp

```
// Делаем этот класс другом, чтобы обеспечить для
// IteratorFacade доступ к основным операциям итератора:
class IteratorFacadeAccess
{
    // Эти определения может использовать только IteratorFacade
    template<typename Derived, typename Value, typename Category,
             typename Reference, typename Distance>
    friend class IteratorFacade;

    // Требуется для всех итераторов:
    template<typename Reference, typename Iterator>
    static Reference dereference(Iterator const& i)
    {
        return i.dereference();
    }
    ...
    // Требуется для двунаправленных итераторов:
    template<typename Iterator>
    static void decrement(Iterator& i)
    {
        return i.decrement();
    }
    // Требуется для итераторов произвольного доступа:
    template<typename Iterator, typename Distance>
    static void advance(Iterator& i, Distance n)
    {
        return i.advance(n);
    }
    ...
};
```

Этот класс предоставляет статические функции-члены для каждой из основных операций итератора, вызывая соответствующие (нестатические) функции-члены предоставленного итератора. Все статические функции-члены являются закрытыми, с доступом только для самого `IteratorFacade`. Таким образом, наш `ListNodeIterator` может сделать `IteratorFacadeAccess` другом и оставить закрытым интерфейс, необходимый для фасада:

```
friend class IteratorFacadeAccess;
```

Адаптеры итераторов

Наш `IteratorFacade` позволяет легко создать *адаптер* итератора, который принимает существующий итератор и возвращает новый итератор, предоставляющий некоторое преобразованное представление базовой последовательности. Например, у нас мог бы существовать контейнер значений `Person`:

inherit/person.hpp

```
struct Person
{
    std::string firstName;
    std::string lastName;
    friend std::ostream& operator<<(std::ostream& strm,
                                    Person const& p)
    {
        return strm << p.lastName << ", " << p.firstName;
    }
};
```

Однако вместо того, чтобы проходить по всем значениям `Person` в контейнере, мы хотим видеть только имена. В этом разделе мы разрабатываем адаптер итератора `ProjectionIterator`, который позволяет нам “спроецировать” значения базового итератора на некоторый указатель на член-данное, например `Person::firstName`.

`ProjectionIterator` представляет собой итератор, определенный в терминах базового итератора (`Iterator`) и типа значения, доступного через итератор (`T`):

inherit/projectioniteratorskel.hpp

```
template<typename Iterator, typename T>
class ProjectionIterator
  :public IteratorFacade <
    ProjectionIterator<Iterator, T>,
    T,
    typename std::iterator_traits<Iterator>::iterator_category,
    T&,
    typename std::iterator_traits<Iterator>::difference_type >
{
    using Base = typename std::iterator_traits<Iterator>::value_type;
    using Distance =
        typename std::iterator_traits<Iterator>::difference_type;
```

```
    Iterator iter;
    T Base::* member;

    friend class IteratorFacadeAccess;
    ... // Реализация основных итераторных
        // операций для IteratorFacade
  public:
    ProjectionIterator(Iterator iter, T Base::* member)
        : iter(iter), member(member) { }
};

template<typename Iterator, typename Base, typename T>
auto project(Iterator iter, T Base::* member)
{
    return ProjectionIterator<Iterator, T>(iter, member);
}
```

Каждый итератор проекции хранит два значения: `iter` — итератор базовой последовательности (значений `Base`) и `member` — указатель на член-данное, описывающий, какой именно член проецируется. С учетом этого мы рассмотрим аргументы шаблона, предоставляемые базовому классу `IteratorFacade`. Первым является сам `ProjectionIterator` (для возможности применения идиомы CRTP). Второй (`T`) и четвертый (`T&`) аргументы представляют собой типы значения и ссылки нашего итератора проекции, определяя его как последовательность значений `T`[4]. Третий и пятый аргументы просто передают категорию и тип разности базового итератора. Таким образом, наш итератор проекции будет входным итератором, когда `Iterator` является входным итератором, двунаправленным итератором, когда `Iterator` является двунаправленным итератором, и так далее. Функция `project()` позволяет легко создать итератор проекции.

Единственной недостающей частью является реализация основных требований к `IteratorFacade`. Наиболее интересна функция `dereference()`, разыменовывающая базовый итератор, а затем выполняющая проекцию:

```
T& dereference() const
{
    return (*iter).*member;
}
```

Остальные операции реализуются в терминах базового итератора:

```
void increment()
{
    ++iter;
}

bool equals(ProjectionIterator const& other) const
{
    return iter == other.iter;
}
```

[4] Для простоты мы вновь считаем, что базовый итератор возвращает ссылку, а не прокси.

```
void decrement()
{
    --iter;
}
```

Для краткости мы опустили определения итераторов произвольного доступа, которые создаются аналогично.

Вот и все! С нашим итератором проекции мы можем вывести имена из вектора, содержащего значения `Person`:

inherit/projectioniterator.cpp

```
#include <vector>
#include <algorithm>
#include <iterator>

int main()
{
    std::vector<Person> authors = {
        {"David", "Vandevoorde"},
        {"Nicolai", "Josuttis"},
        {"Douglas", "Gregor"}
    };
    std::copy(project(authors.begin(), &Person::firstName),
              project(authors.end(), &Person::firstName),
              std::ostream_iterator<std::string>(std::cout, "\n"));
}
```

Вывод программы имеет следующий вид:

```
David
Nicolai
Douglas
```

Проектный шаблон “Фасад” особенно полезен для создания новых типов, которые соответствуют некоторому определенному интерфейсу. Новые типы должны предоставлять фасаду только несколько основных операций (от 3 до 6 для нашего фасада итератора), а он заботится о предоставлении полного и корректного открытого интерфейса, используя комбинацию CRTP и метода Бартона–Нэкмана.

21.3. Миксины

Рассмотрим простой класс `Polygon`, состоящий из последовательности точек:

```
class Point
{
  public:
    double x, y;
    Point() : x(0.0), y(0.0) { }
    Point(double x, double y) : x(x), y(y) { }
};
```

```
class Polygon
{
  private:
    std::vector<Point> points;
  public:
    ... // Открытые операции
};
```

Этот класс `Polygon` будет более полезным, если пользователь сможет расширить набор сведений, связанных с каждой точкой `Point`, чтобы иметь возможность указать информацию приложения, такую как цвет каждой точки, или. возможно, связать с каждой точкой метку. Один из вариантов сделать такое расширение возможным — параметризовать `Polygon` типом точки:

```
template<typename P>
class Polygon
{
  private:
    std::vector<P> points;
  public:
    ... // Открытые операции
};
```

Пользователи могут создавать свой собственный “точкообразный” тип данных, предоставляющий тот же интерфейс, что и `Point`, но включающий и другие данные приложения с помощью наследования:

```
class LabeledPoint : public Point
{
  public:
    std::string label;
    LabeledPoint() : Point(), label("") { }
    LabeledPoint(double x, double y) : Point(x, y), label("") { }
};
```

Эта реализация имеет свои недостатки. С одной стороны она требует, чтобы тип `Point` был доступен пользователю, так, чтобы пользователь мог его наследовать. Кроме того, автор `LabeledPoint` должен быть осторожным и предоставить точно такой же интерфейс, как и `Point` (например, наследованием или предоставлением всех тех же конструкторов, что и у `Point`), иначе `LabeledPoint` не сможет работать с `Polygon`. Это ограничение становится более проблематичным, если `Point` изменяется от одной версии шаблона `Polygon` к другой: добавление нового конструктора `Point` может потребовать обновления каждого производного класса.

Миксины (mixins) обеспечивают альтернативный способ настроить поведение типа без наследования от него. Миксины, по существу, инвертируют нормальное направление наследования, потому что новые классы являются “подмешанными” (mixed in) в иерархию наследования как базовые классы шаблона класса, а не создаются как новые производные классы. Этот подход разрешает введение новых членов-данных и других операций, не требуя какого-либо дублирования интерфейса.

Шаблон класса, который поддерживает миксины, обычно принимает произвольное количество дополнительных классов, от которых он наследуется:

```
template<typename... Mixins>
class Point : public Mixins...
{
  public:
    double x, y;
    Point() : Mixins()..., x(0.0), y(0.0) { }
    Point(double x, double y) : Mixins()..., x(x), y(y) { }
};
```

Теперь мы можем “подмешать” базовый класс с меткой, чтобы получить `LabeledPoint`:

```
class Label
{
  public:
    std::string label;
    Label() : label("") { }
};
using LabeledPoint = Point<Label>;
```

или даже добавить несколько базовых классов:

```
class Color
{
  public:
    unsigned char red = 0, green = 0, blue = 0;
};
using MyPoint = Point<Label, Color>;
```

Этот основанный на миксинах класс `Point` позволяет легко предоставить дополнительную информацию к классу `Point` без изменения его интерфейса, так что `Polygon` становится проще использовать и развивать. Пользователи должны только применять неявное преобразование из специализации `Point` в соответствующий класс миксина (такой как `Label` или `Color`, показанные выше) для доступа к данным или интерфейсу. Кроме того, класс `Point` может быть даже полностью скрытым с помощью предоставления миксинов шаблону класса `Polygon`:

```
template<typename... Mixins>
class Polygon
{
  private:
    std::vector<Point<Mixins...>> points;
  public:
    ... // Открытые операции
};
```

Миксины полезны в тех случаях, когда шаблон требует некоторого небольшого уровня настройки — например, декорирование внутренне хранимых объектов пользовательскими данными — без необходимости требовать от библиотеки открытия и документирования этих внутренних типов данных и их интерфейсов.

21.3.1. Странные миксины

Миксины могут быть еще более мощным средством при объединении их со странно рекурсивным шаблоном проектирования (CRTP), описанным в разделе 21.2. Здесь каждый из миксинов на самом деле является шаблоном класса,

который будет предоставлен с типом производного класса, что обеспечивает дополнительные настройки этого производного класса. Версия класса Point на основе миксина с применением идиомы CRTP будет выглядеть следующим образом:

```
template<template<typename>... Mixins>
class Point : public Mixins<Point>...
{
  public:
    double x, y;
    Point() : Mixins<Point>()..., x(0.0), y(0.0) { }
    Point(double x, double y) : Mixins<Point>()..., x(x), y(y) { }
};
```

Эта формулировка требует немного больше работы для каждого класса, который будет "подмешан", поэтому такие классы, как Label и Color, должны стать шаблонами классов. Однако теперь миксины могут адаптировать свое поведение для определенного экземпляра производного класса, к которому они "подмешиваются". Например, можно смешать рассматривавшийся ранее шаблон ObjectCounter с Point, чтобы подсчитывать количество точек, создаваемых Polygon, или объединять миксины с другими миксинами, специфичными для данного приложения.

21.3.2. Параметризованная виртуальность

Миксины также позволяют косвенно параметризовать такие атрибуты производного класса, как виртуальность функции-члена. Простой пример демонстрирует эту весьма удивительную технику:

inherit/virtual.cpp

```
#include <iostream>

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

template<typename... Mixins>
class Base : public Mixins...
{
  public:
    // Виртуальность foo() зависит от ее объявления
    // (если таковое имеется) в миксинах базовых классов...
    void foo()
    {
        std::cout << "Base::foo()" << '\n';
    }
};
```

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

int main()
{
    Base<NotVirtual>* p1 = new Derived<NotVirtual>;
    p1->foo();         // Вызов Base::foo()
    Base<Virtual>* p2 = new Derived<Virtual>;
    p2->foo();         // Вызов Derived::foo()
}
```

Эта методика может служить инструментом для проектирования шаблона класса, используемого как для инстанцирования конкретных классов, так и для расширения с применением наследования. Однако для получения класса, который оказывается хорошим базовым классом для более специализированной функциональности, редко бывает достаточно всего лишь придать виртуальность некоторым функциям-членам. Метод разработки такого вида требует более фундаментальных проектных решений. Поэтому обычно более практичной оказывается разработка двух различных инструментов (иерархии классов или шаблонов классов), чем попытки интегрировать их в единую иерархию шаблонов.

21.4. Именованные аргументы шаблона

Иногда различные методы работы с шаблонами приводят к тому, что шаблон содержит весьма значительное число разных типовых параметров. Конечно, как правило, многие из них имеют вполне приемлемые значения по умолчанию. Естественный способ определения такого шаблона класса может выглядеть так, как показано ниже.

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

Вероятно, такой шаблон чаще всего будет использоваться с аргументами по умолчанию с применением синтаксиса BreadSlicer<>. Однако, если некоторый аргумент имеет значение не по умолчанию, все предшествующие ему аргументы должны быть явно указаны (даже если они используют значения по умолчанию).

Понятно, что было бы гораздо привлекательнее использовать конструкцию `BreadSlicer<Policy3 = Custom>`, чем стандартную, имеющую вид `BreadSlicer<DefaultPolicy1,DefaultPolicy2,Custom>`. Ниже рассматривается методика, позволяющая использовать синтаксис, очень похожий на описанный[5].

Наш метод заключается в размещении значений типа по умолчанию в базовом классе и в переопределении некоторых из них в процессе наследования. Вместо непосредственного указания аргументов типа предоставим их через вспомогательные классы. Например, можно написать `BreadSlicer<Policy3_is<Custom>>`. Поскольку каждый аргумент шаблона может описывать любую из стратегий, значения по умолчанию не могут быть различными. Иными словами, на верхнем уровне все параметры шаблона эквивалентны:

```
template<typename PolicySetter1 = DefaultPolicyArgs,
         typename PolicySetter2 = DefaultPolicyArgs,
         typename PolicySetter3 = DefaultPolicyArgs,
         typename PolicySetter4 = DefaultPolicyArgs>
class BreadSlicer
{
    using Policies = PolicySelector<PolicySetter1, PolicySetter2,
                                    PolicySetter3, PolicySetter4>;
    // Используем Policies::P1, Policies::P2, ...
    // для обращения к различным стратегиям
    ...
};
```

После этого остается только одна проблема — написать шаблон `PolicySelector`. Он должен объединить различные аргументы шаблона в единый тип, который перекрывает используемые по умолчанию члены-псевдонимы типа. Такое объединение может быть достигнуто с использованием наследования:

```
// PolicySelector<A,B,C,D> создает A,B,C,D в качестве
// базовых классов. Discriminator<> даже позволяет иметь
// несколько одинаковых базовых классов
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>
{
};
```

Обратите внимание на использование промежуточного шаблона `Discriminator`. Он необходим для того, чтобы можно было иметь одинаковые типы `Setter`.

[5] Обратите внимание на то, что подобное расширение языка для аргументов вызова функции было предложено (и отклонено) в процессе стандартизации языка C++ еще раньше (более подробно об этом говорится в разделе 17.4).

(Иметь несколько непосредственных базовых классов одного и того же типа нельзя; обойти это ограничение можно с помощью опосредованного наследования.)

Как обещали ранее, соберем в базовом классе все значения по умолчанию.

```
// Именуем стратегии по умолчанию как P1, P2, P3, P4
class DefaultPolicies
{
  public:
    using P1 = DefaultPolicy1;
    using P2 = DefaultPolicy2;
    using P3 = DefaultPolicy3;
    using P4 = DefaultPolicy4;
};
```

Однако мы должны быть внимательны и избегать неоднозначности при многократном наследовании от этого базового класса, т.е. базовый класс должен наследоваться виртуально:

```
// Класс для стратегий по умолчанию позволяет избежать
// неоднозначности при помощи виртуального наследования
class DefaultPolicyArgs : virtual public DefaultPolicies
{
};
```

Наконец, нужно написать ряд шаблонов для перекрытия значений стратегий, заданных по умолчанию:

```
template<typename Policy>
class Policy1_is : virtual public DefaultPolicies
{
  public:
    using P1 = Policy; // Перекрытие шаблона типа
};

template<typename Policy>
class Policy2_is : virtual public DefaultPolicies
{
  public:
    using P2 = Policy; // Перекрытие шаблона типа
};

template<typename Policy>
class Policy3_is : virtual public DefaultPolicies
{
  public:
    using P3 = Policy; // Перекрытие шаблона типа
};

template<typename Policy>
class Policy4_is : virtual public DefaultPolicies
{
  public:
    using P4 = Policy; // Перекрытие шаблона типа
};
```

Теперь мы можем достичь нашей цели. Рассмотрим конкретный пример и инстанцируем `BreadSlicer<>` следующим образом:

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

Для этого `BreadSlicer<>` тип `Policies` определен как

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

С помощью шаблона класса `Discriminator<>` в результате будет получена иерархия, в которой все аргументы шаблона являются базовыми классами (рис. 21.4). Важное замечание: все эти базовые классы имеют один и тот же виртуальный базовый класс `DefaultPolicies`, который определяет заданные по умолчанию типы для `P1`, `P2`, `P3` и `P4`. Однако `P3` переопределен в одном из порожденных классов, а именно — в классе `Policy3_is<>`. Согласно так называемому *правилу доминирования* (domination rule), это определение скрывает определение базового класса. Таким образом, здесь *нет* неоднозначности[6].

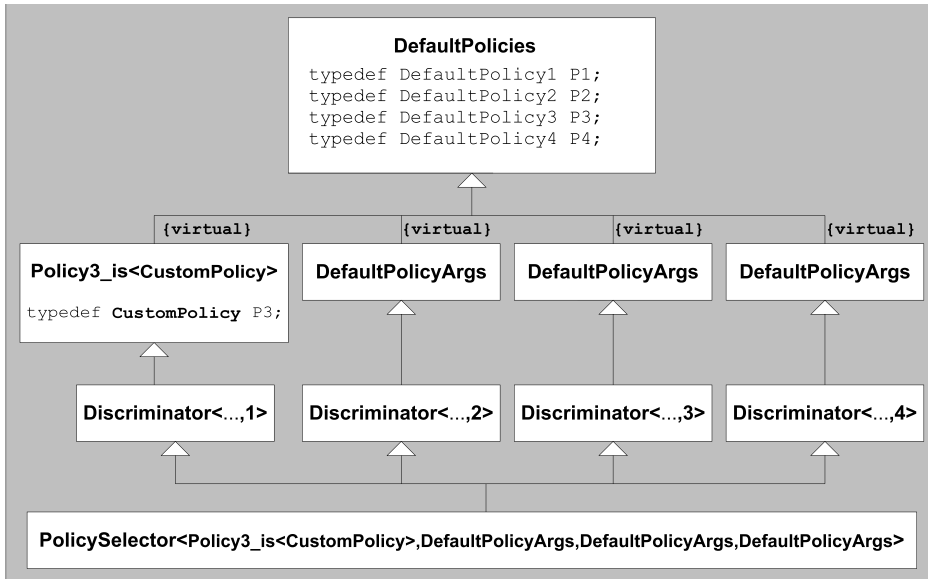

Рис. 21.4. Иерархия типов `BreadSlicer<>::Policies`

Внутри шаблона `BreadSlicer` можно обращаться к четырем стратегиям, используя для этого квалифицированные имена наподобие `Policies::P3`. Например:

[6] Определение правила доминирования можно найти в разделе 10.2/6 первого Стандарта C++ [25], а также в [39], раздел 10.1.1.

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

Полный исходный текст можно найти в файле `inherit/namedtmpl.cpp`.

Здесь разработана методика для четырех параметров шаблона, но очевидно, что эта методика применима для любого разумного количества таких параметров. Обратите внимание на то, что при этом нигде не было реального инстанцирования объекта вспомогательного класса, содержащего виртуальные базовые классы. Следовательно, тот факт, что они являются виртуальными базовыми классами, не влияет на производительность или потребление памяти.

21.5. Заключительные замечания

Билл Гиббонс (Bill Gibbons) был основным инициатором введения оптимизации пустого базового класса в язык программирования C++, а Натан Майерс (Nathan Myers) предложил шаблон, подобный нашему `BaseMemberPair`, для получения максимальной пользы от применения данной оптимизации. В библиотеке Boost имеется значительно более сложный шаблон `compressed_pair`, который решает ряд упомянутых в данной главе проблем для шаблона `MyClass` и может использоваться вместо разработанного нами шаблона `BaseMemberPair`.

Метод CRTP используется еще с 1991 года. Первым этот метод формально описал Джеймс Коплин (James Coplien) [32]. С тех пор было опубликовано множество разнообразных применений CRTP. Однако иногда CRTP ошибочно применяют к *параметризованному наследованию*. Как было показано, CRTP не требует, чтобы наследование было параметризовано, а многие формы параметризованного наследования не согласуются с CRTP. Кроме того, CRTP иногда путают с методом Бартона–Нэкмана (см. раздел 21.2.1), поскольку Бартон и Нэкман часто использовали CRTP в комбинации с введением дружественных имен (каковое является важным компонентом метода Бартона–Нэкмана). Наше использование CRTP с методом Бартона–Нэкмана для реализации операторов следует тому же основному подходу, что и библиотека Boost.Operators [17], которая предоставляет обширный набор определений операторов. Аналогично наше рассмотрение фасадов итераторов следует библиотеке Boost.Iterator [15], которая обеспечивает богатый, соответствующий требованиям стандартной библиотеки интерфейс итераторов для производного типа, который предоставляет несколько основных операций итератора (равенство, разыменование, перемещение), а также решает сложные вопросы с участием прокси-итераторов (которые в книге не

рассматривались). Наш пример ObjectCounter практически идентичен методу, разработанному Скоттом Мейерсом (Scott Meyers) [51].

Понятие миксинов в объектно-ориентированном программировании появилось не позже 1986 года [54] как способ внести небольшие фрагменты функциональности в объектно-ориентированный класс. Использование шаблонов C++ для миксинов приобрело популярность вскоре после публикации первого стандарта C++, когда в статьях [61] и [38] были описаны подходы, широко используемые сегодня для миксинов. С тех пор они стали популярным методом в проектировании библиотек C++.

Именованные аргументы шаблона используются для упрощения некоторых шаблонов классов в библиотеке Boost, которая применяет метапрограммирование для создания типа со свойствами, схожими с нашим PolicySelector (но без использования виртуального наследования). Более простая альтернатива, представленная здесь, была разработана одним из авторов книги — Дэвидом Вандевурдом.