

Если производный класс наследуется от невиртуального базового класса несколькими путями, то он наследует по одному объекту базового класса для каждого экземпляра базового класса. В некоторых случаях так и нужно, но чаще наличие нескольких экземпляров базового класса приводит к проблемам.

Теперь рассмотрим множественное наследование с виртуальными базовыми классами. Класс становится виртуальным базовым классом, когда в производном классе при описании наследования используется ключевое слово `virtual`:

```
class marketing : public virtual reality { ... };
```

Главное отличие и причина применения виртуальных базовых классов заключается в том, что класс, порождаемый от одного или более экземпляров виртуального базового класса, наследует только один объект базового класса. Реализация этого свойства приводит к следующим требованиям.

- В производном классе с непрямым виртуальным базовым классом конструкторы должны непосредственно вызывать конструкторы непрямых базовых классов, что недопустимо для непрямых виртуальных базовых классов.
- Неоднозначность имен разрешается в соответствии с правилами доминирования.

Как видите, множественное наследование может приводить к сложностям в программировании. Правда, эти сложности возникают тогда, когда производный класс наследуется несколькими путями от одного и того же базового класса. Если не ввязываться в такие ситуации, то остается лишь при необходимости уточнять унаследованные имена.

Шаблоны классов

Наследование (открытое, закрытое и защищенное) и включение не всегда являются решением, позволяющим повторно использовать код. Рассмотрим, например, класс `Stack` (см. главу 10) и класс `Queue` (см. главу 12). Это примеры *контейнерных классов*, содержащих другие типы объектов или данных. Класс `Stack` из главы 10, например, содержит значения `unsigned long`. Легко можно создать стек для хранения значений `double` или объектов `string`. Код будет аналогичным, за исключением типов хранимых объектов. Но вместо того чтобы определять новые классы, хорошо было бы определить стек в общей (не зависящей от типа) форме и задавать конкретные типы как параметры класса. Тогда один и тот же общий код можно будет использовать для создания стеков разных типов величин. В главе 10 в примере `Stack` в качестве первого шага в достижении этой цели используется конструкция `typedef`. Но такому подходу присуща пара недостатков. Во-первых, при каждом изменении типа придется редактировать заголовочный файл. Во-вторых, этот прием позволяет создать лишь один вид стека на программу. Ведь `typedef` не может определять два разных типа одновременно, и поэтому невозможно так создать одновременно, скажем, стек значений `int` и стек объектов `string`.

Шаблоны классов в C++ предлагают более подходящий способ создания обобщенных определений класса. (Изначально язык C++ не поддерживал шаблоны, а с момента появления они постоянно развиваются. Поэтому если используется устаревший компилятор, могут поддерживаться не все рассматриваемые здесь возможности.) Шаблоны предоставляют *параметризованные* типы – т.е. при создании класса или функции можно передать имя типа в качестве аргумента. Передав, к примеру, в шаблон `Queue` имя типа `int`, можно указать компилятору создать класс `Queue` для хранения в очереди целых значений.

Библиотека C++ содержит несколько шаблонов классов. Ранее в этой главе рассматривался шаблон класса `valarray`, а в главе 4 были описаны шаблонные классы `vector` и `array`. Стандартная библиотека шаблонов (Standard Template Library – STL) C++, частично рассматриваемая в главе 16, предлагает мощные и гибкие реализации шаблонов для нескольких контейнерных классов. В этой главе мы ознакомимся с построением более простых конструкций.

Определение шаблона класса

В качестве модели для построения шаблона воспользуемся классом `Stack` из главы 10. Вот исходное определение класса:

```
typedef unsigned long Item;

class Stack
{
private:
    enum {MAX = 10};                      // константа, характерная для класса
    Item items[MAX];                     // содержит элементы стека
    int top;                             // индекс вершины стека

public:
    Stack();
    bool isempty() const;
    bool isfull() const;

    // push() возвращает false, если стек полон, и true – в противном случае
    bool push(const Item & item);        // добавляет элемент в стек

    // pop() возвращает false, если стек пуст, и true – в противном случае
    bool pop(Item & item);              // выталкивает элемент с вершины стека
};
```

При построении шаблона определение класса `Stack` заменяется определением шаблона, а функции-члены класса – функциями-членами шаблона. Как и для шаблонных функций, шаблонный класс предваряется следующим кодом:

```
template <class Тип>
```

Ключевое слово `template` сообщает компилятору, что далее следует определение шаблона. Часть кода в угловых скобках аналогична списку аргументов в функции. Можно считать, что ключевое слово `class` служит именем типа для переменной, которая получает тип как значение, а `Тип` является именем этой переменной.

Слово `class` не означает, что `Тип` должно быть классом; это означает только, что `Тип` служит в качестве спецификатора обобщенного типа, который будет заменен реальным типом при использовании шаблона. Последние реализации C++ позволяют применять вместо `class` более точное ключевое слово `typename`:

```
template <typename Тип>           // новый вариант
```

Вместо `Тип` можно вписать свое имя обобщенного типа; правила именования здесь такие же, что и для любого другого идентификатора. Обычно используют идентификаторы `T` и `Type`; мы будем применять `Type`. При вызове шаблона параметр `Тип` заменяется конкретным типом вроде `int` или `string`. Внутри определения шаблона имя обобщенного типа можно использовать для указания типа, который будет храниться в стеке. В случае класса `Stack` нужно указывать `Type` везде, где в старом определении применялся идентификатор `Item` из конструкции `typedef`. Например:

```
Item items[MAX];                  // содержит элементы стека
```

станет выглядеть как

```
Type items[MAX]; // содержит элементы стека
```

Аналогично можно заменить методы исходного класса функциями-членами шаблона. Каждая функция должна предваряться таким же объявлением шаблона:

```
template <class Type>
```

Здесь также необходимо заменить идентификатор `Item` из конструкции `typedef` именем обобщенного типа `Type`. Кроме того, квалификатор класса `Stack::` нужно заменить вариантом `Stack<Type>::`. Например:

```
bool Stack::push(const Item & item)
{
    ...
}
```

преобразуется в:

```
template <class Type> // или template <typename Type>
bool Stack<Type>::push(const Type & item)
{
    ...
}
```

Если внутри определения класса определить метод (встроенное определение), можно опустить преамбулу шаблона и квалификатор класса.

В листинге 14.13 приведены комбинированные шаблоны класса и функций-членов. Важно понимать, что эти шаблоны не являются определениями классов и функций-членов. Это скорее указания компилятору C++, как сгенерировать определения класса и функций-членов. Конкретная актуализация шаблона – например, класс стека для управления объектами `string` – называется *созданием экземпляра* или *специализацией*. Шаблонные функции-члены нельзя размещать в отдельном файле реализации. (Одно время в стандарте языка существовало ключевое слово `export`, которое позволяло такой вынос в отдельный файл реализации. Однако оно не было учтено в очень многих реализациях. В C++11 это слово уже не входит в стандарт, но зарезервировано для возможного дальнейшего использования.) Поскольку шаблоны не являются функциями, их нельзя компилировать отдельно. Шаблоны необходимо применять совместно с запросами на создание экземпляров шаблонов. Проще всего это сделать, поместив всю информацию о шаблонах в заголовочный файл и включив этот заголовочный файл в файл, использующий шаблоны.

Листинг 14.13. stacktp.h

```
// stacktp.h -- шаблон стека
#ifndef STACKTP_H_
#define STACKTP_H_
template <class Type>

class Stack
{
private:
    enum {MAX = 10}; // константа, специфичная для класса
    Type items[MAX]; // содержит элементы стека
    int top; // индекс вершины стека
public:
    Stack();
    bool isempty();
```

```

bool isfull();
bool push(const Type & item);           // добавление item в стек
bool pop(Type & item);                 // выталкивание из стека в item
};

template <class Type>
Stack<Type>::Stack()
{
    top = 0;
}

template <class Type>
bool Stack<Type>::isempty()
{
    return top == 0;
}

template <class Type>
bool Stack<Type>::isfull()
{
    return top == MAX;
}

template <class Type>
bool Stack<Type>::push(const Type & item)
{
    if (top < MAX)
    {
        items[top++] = item;
        return true;
    }
    else
        return false;
}

template <class Type>
bool Stack<Type>::pop(Type & item)
{
    if (top > 0)
    {
        item = items[--top];
        return true;
    }
    else
        return false;
}
#endif

```

Использование шаблонного класса

Просто включение шаблона в программу не генерирует шаблонный класс – необходимо запросить создание экземпляра. Для этого потребуется объявить объект с типом шаблонного класса и заменить имя обобщенного типа конкретным типом. Например, ниже показано создание двух стеков: одного для хранения значений `int`, а другого – для объектов `string`.

```

Stack<int> kernels;                      // создание стека для значений int
Stack<string> colonels;                  // создание стека для объектов string

```

Встретив эти два определения, компилятор на основе шаблона `Stack<Type>` сгенерирует два различных объявления класса и два разных набора методов класса. Объявление класса `Stack<int>` заменит все `Type` на `int`, а объявление класса `Stack<string>` заменит все `Type` на `string`. Конечно, используемые алгоритмы должны согласоваться с типами. К примеру, класс `Stack` предполагает, что возможно присваивание одних элементов другим. Это присваивание справедливо для базовых типов, структур и классов (если не сделать операцию присваивания закрытой), но не для массивов.

Идентификаторы обобщенных типов, такие как `Type` в нашем примере, называются *параметрами типа* — т.е. они действуют примерно как переменные, но вместо присваивания числового значения параметру типа присваивается тип. Поэтому в объявлении `kernels` параметр `Type` имеет значение `int`.

Учтите, что требуемый тип необходимо указать явно. В этом заключается отличие от обычных шаблонных функций — в них компилятор может использовать типы аргументов для определения вида функции, которую нужно сгенерировать:

```
template <class T>
void simple(T t) { cout << t << '\n'; }
...
simple(2);           // генерирует void simple(int)
simple("two");       // генерирует void simple(const char *)
```

В листинге 14.14 приведена первоначальная программа тестирования работы стека (см. листинг 10.12), но приспособленная для использования строкового представления идентификаторов заказов вместо значений `unsigned long`.

Листинг 14.14. stacktem.cpp

```
// stacktem.cpp -- тестирование шаблонного класса стека
#include <iostream>
#include <string>
#include <cctype>
#include "stacktp.h"
using std::cin;
using std::cout;
int main()
{
    Stack<std::string> st;                                // создание пустого стека
    char ch;
    std::string po;
    cout << "Please enter A to add a purchase order, \n" // A - добавить заказ,
        << "P to process a PO, or Q to quit. \n";      // P - обработать заказ, Q - выйти
    while (cin >> ch && std::toupper(ch) != 'Q')
    {
        while (cin.get() != '\n')
            continue;
        if (!std::isalpha(ch))
        {
            cout << '\a';
            continue;
        }
        switch(ch)
        {
            case 'A':
            case 'a': cout << "Enter a PO number to add: ";
                        // Ввод номера добавляемого заказа
```

```

        cin >> po;
        if (st.isfull())
            cout << "stack already full\n";           // стек уже полон
        else
            st.push(po);
            break;
    case 'P':
    case 'p': if (st.isempty())
        cout << "stack already empty\n";           // стек уже пуст
    , else {
        st.pop(po);
        cout << "PO #" << po << " popped\n";      // заказ извлечен
        break;
    }
}
cout << "Please enter A to add a purchase order,\n"
    << "P to process a PO, or Q to quit.\n";
}
cout << "Bye\n";
return 0;
}

```

Ниже приведен пример запуска программы из листинга 14.14:

Please enter A to add a purchase order,
P to process a PO, or Q to quit.

A

Enter a PO number to add: **red911porsche**

Please enter A to add a purchase order,
P to process a PO, or Q to quit.

A

Enter a PO number to add: **blueR8audi**

Please enter A to add a purchase order,
P to process a PO, or Q to quit.

A

Enter a PO number to add: **silver747boeing**

Please enter A to add a purchase order,
P to process a PO, or Q to quit.

P

PO #silver747boeing popped

Please enter A to add a purchase order,
P to process a PO, or Q to quit.

P

PO #blueR8audi popped

Please enter A to add a purchase order,
P to process a PO, or Q to quit.

P

PO #red911porsche popped

Please enter A to add a purchase order,
P to process a PO, or Q to quit.

Q

stack already empty

Please enter A to add a purchase order,
P to process a PO, or Q to quit.

Q

Bye

Более внимательный взгляд на шаблонные классы

В качестве типа для шаблона класса `Stack<Type>` можно использовать встроенный тип или объект класса. А как насчет указателей? Например, можно ли применить в листинге 14.14 не объект `string`, а указатель на `char`? В конце концов, такие указатели являются встроенным средством для работы со строками в стиле С. Ответ таков: конечно, можно создать стек указателей, но он будет плохо работать без существенной переделки программ. Компилятор может создать какой угодно класс, однако задача программиста — правильно его использовать. Сначала посмотрим, почему такой стек указателей будет плохо работать с кодом из листинга 14.14, а затем рассмотрим пример полезного применения стека указателей.

Некорректное использование стека указателей

Давайте кратко рассмотрим три простых, но неудачных попытки адаптации листинга 14.14 к использованию стека указателей. Из этих примеров необходимо извлечь урок, чтобы в дальнейшем при создании шаблонов не действовать вслепую. Все три примера начинаются с совершенно допустимого вызова шаблона `Stack<Type>`:

```
Stack<char *> st; // создание стека указателей на символы
```

Вариант 1: оператор

```
string po;
```

из листинга 14.14 меняется на

```
char * po;
```

Смысл в том, чтобы для ввода данных с клавиатуры использовать указатель на `char` вместо объекта `string`. Этот подход ошибочен с самого начала — ведь одно только создание указателя не выделяет память для хранения входных строк. (Программа скомпилируется нормально, но, скорее всего, завершится аварийно, когда `cin` попытается сохранить введенные данные в неподходящем месте.)

Вариант 2: оператор

```
string po;
```

заменяется на

```
char po[40];
```

Здесь выделяется память для входной строки, а переменная `po` имеет тип `char *`, и поэтому ее можно поместить в стек. Однако массив ведет себя совершенно не так, как нужно в методе `pop()`:

```
template <class Type>
bool Stack<Type>::pop(Type & item)
{
    if (top > 0)
    {
        item = items[--top];
        return true;
    }
    else
        return false;
}
```

Во-первых, ссылочная переменная `item` должна ссылаться на некоторого вида `lvalue`, но не на имя массива. Во-вторых, предполагается, что переменной `item` можно

присваивать значения. Даже если бы переменная `item` могла ссылаться на массив, невозможно присвоить значение имени массива. Так что этот способ тоже не годится.

Вариант 3: оператор

```
string po;
```

заменяется на

```
char * po = new char[40];
```

Здесь выделяется память для входной строки, а `po` является переменной и поэтому совместима с кодом метода `pop()`. Однако здесь мы сталкиваемся с наиболее фундаментальной проблемой: имеется только одна переменная `po`, и она всегда указывает на одно и тоже место в памяти. Правда, содержимое памяти меняется при каждом чтении новой строки, но каждая операция заталкивания помещает в стек один и тот же адрес. Поэтому при выталкивании данных из стека мы всегда будем получать один и тот же адрес, и он всегда будет указывать на последнюю строку, прочитанную и сохраненную в памяти. Такой стек не сохраняет отдельно каждую новую строку по мере их ввода и поэтому бесполезен.

Корректное использование стека указателей

Один из способов применения стека указателей – создание в вызывающей программе массива указателей, где все указатели указывают на разные строки. Помещение таких указателей в стек имеет смысл, т.к. они ссылаются на разные строки. Обратите внимание, что создание различных указателей – обязанность вызывающей программы, а не стека. Стек должен просто манипулировать готовыми указателями, а не создавать их.

Предположим, что нужно смоделировать следующую ситуацию. Секретарь привез преподавателю тележку с объемными курсовыми работами студентов. Если входной ящик преподавателя пуст, он берет из тележки верхнюю работу и кладет во входной ящик. Если входной ящик заполнен, преподаватель берет из него верхнюю работу, проверяет ее и кладет в выходной ящик. Если входной ящик заполнен частично, преподаватель может проверить верхнюю работу из входного ящика, а может взять верхнюю работу из тележки и положить во входной ящик. Чтобы решить, как поступить в каждом таком случае, он просто подбрасывает монетку. Попытаемся исследовать влияние его действий на первоначальный порядок курсовых работ.

Описанную ситуацию можно смоделировать с помощью массива указателей на строки, представляющие курсовые работы в тележке. Каждая строка содержит имя студента, написавшего работу. Для представления входного ящика можно использовать стек, а для представления выходного ящика – еще один массив указателей. Добавление работы во входной ящик можно представить заталкиванием указателя из входного массива в стек, а обработку папки – выталкиванием элемента из стека и добавлением его в выходной ящик.

Учитывая важность исследования всех аспектов данной задачи, будет полезно иметь возможность опробовать разные размеры стека. В листинге 14.15 класс `Stack<Type>` слегка переопределен так, чтобы конструктор `Stack` принимал размер стека в качестве аргумента. Это приводит к внутреннему использованию динамического массива, поэтому классу теперь требуется деструктор, конструктор копирования и операция присваивания. Кроме того, определение сокращает объем кода, т.к. некоторые методы встроены в код определения.

Листинг 14.15. stcktpl.h

```

// stcktpl.h -- модифицированный шаблон Stack
#ifndef STCKTPL_H_
#define STCKTPL_H_

template <class Type>
class Stack
{
private:
    enum {SIZE = 10};                      // размер по умолчанию
    int stacksize;
    Type * items;                         // хранит элементы стека
    int top;                             // индекс вершины стека
public:
    explicit Stack(int ss = SIZE);
    Stack(const Stack & st);
    ~Stack() { delete [] items; }
    bool isempty() { return top == 0; }
    bool isfull() { return top == stacksize; }
    bool push(const Type & item);        // добавление item в стек
    bool pop(Type & item);              // выталкивание верхнего элемента в item
    Stack & operator=(const Stack & st);
};

template <class Type>
Stack<Type>::Stack(int ss) : stacksize(ss), top(0)
{
    items = new Type [stacksize];
}

template <class Type>
Stack<Type>::Stack(const Stack & st)
{
    stacksize = st.stacksize;
    top = st.top;
    items = new Type [stacksize];
    for (int i = 0; i < top; i++)
        items[i] = st.items[i];
}

template <class Type>
bool Stack<Type>::push(const Type & item)
{
    if (top < stacksize)
    {
        items[top++] = item;
        return true;
    }
    else
        return false;
}

template <class Type>
bool Stack<Type>::pop(Type & item)
{
    if (top > 0)
    {
        item = items[--top];
        return true;
    }
}

```

774 Глава 14

```
    else
        return false;
}

template <class Type>
Stack<Type> & Stack<Type>::operator=(const Stack<Type> & st)
{
    if (this == &st)
        return *this;
    delete [] items;
    stacksize = st.stacksize;
    top = st.top;
    items = new Type [stacksize];
    for (int i = 0; i < top; i++)
        items[i] = st.items[i];
    return *this;
}
#endif
```

Обратите внимание, что прототип объявляет тип, возвращаемый функцией операции присваивания, как ссылку на Stack, а само определение шаблонной функции задает тип как Stack<Type>. Первое объявление является сокращением для второго, но может использоваться только внутри области видимости класса. То есть можно применять тип Stack внутри определения шаблонов и шаблонных функций, а за пределами класса – например, при указании возвращаемых типов и использовании операции разрешения контекста – необходима полная форма Stack<Type>.

Программа в листинге 14.16 использует новый шаблон стека для моделирования действий преподавателя. Как и в предыдущих примерах, для генерации случайных чисел в ней используются функции rand(), srand() и time(). Случайно сгенерированные 0 и 1 моделируют подбрасывание монеты.

Листинг 14.16. stkptr1.cpp

```
// stkptr1.cpp -- тестирование стека указателей
#include <iostream>
#include <cstdlib>                                // для rand(), srand()
#include <ctime>                                  // для time()
#include "stcktp1.h"
const int Num = 10;

int main()
{
    std::srand(std::time(0));                      // randomизация rand()
    std::cout << "Please enter stack size: ";      // ввод размера стека
    int stacksize;
    std::cin >> stacksize;

    // Создание пустого стека размером stacksize
    Stack<const char *> st(stacksize);

    // Входной ящик
    const char * in[Num] = {
        " 1: Hank Gilgamesh", " 2: Kiki Ishtar",
        " 3: Betty Rocker", " 4: Ian Flagranti",
        " 5: Wolfgang Kibble", " 6: Portia Koop",
        " 7: Joy Almondo", " 8: Xaverie Paprika",
        " 9: Juan Moore", "10: Misha Mache"
    };
}
```

```

// Выходной ящик
const char * out[Num];
int processed = 0;
int nextin = 0;
while (processed < Num)
{
    if (st.isempty())
        st.push(in[nextin++]);
    else if (st.isfull())
        st.pop(out[processed++]);
    else if (std::rand() % 2 && nextin < Num) // шансы 50 на 50
        st.push(in[nextin++]);
    else
        st.pop(out[processed++]);
}
for (int i = 0; i < Num; i++)
    std::cout << out[i] << std::endl;
std::cout << "Bye\n";
return 0;
}

```

Ниже приведены два примера запуска программы из листинга 14.16 (из-за случайного выбора конечный порядок работ может существенно изменяться даже при одинаковом размере стека):

```

Please enter stack size: 5
2: Kiki Ishtar
1: Hank Gilgamesh
3: Betty Rocker
5: Wolfgang Kibble
4: Ian Flagranti
7: Joy Almondo
9: Juan Moore
8: Xaverie Paprika
6: Portia Koop
10: Mishá Mache
Bye

```

```

Please enter stack size: 5
3: Betty Rocker
5: Wolfgang Kibble
6: Portia Koop
4: Ian Flagranti
8: Xaverie Paprika
9: Juan Moore
10: Mishá Mache
7: Joy Almondo
2: Kiki Ishtar
1: Hank Gilgamesh
Bye

```

Замечания по программе

Строки в программе, представленной в листинге 14.16, никуда не перемещаются. При заталкивании строки в стек просто создается новый указатель на уже существующую строку. То есть создается указатель с адресом существующей строки. При выталкивании строки из стека этот адрес копируется в выходной массив.

В программе используется тип `const char *`, т.к. массив указателей инициализируется набором строковых констант.

Как воздействует деструктор стека на строки? Никак. Конструктор класса использует операцию `new` для создания массива, содержащего указатели. Деструктор класса уничтожает этот массив, а не строки, на которые ссылаются элементы массива.

Пример шаблона массива и нетипизированные аргументы

Шаблоны часто используются для контейнерных классов, поскольку идея параметров типа удачно сочетается с идеей общего способа хранения для различных типов. На самом деле стремление предоставить повторно используемый код для контейнерных классов и было главной причиной введения шаблонов. Рассмотрим другой пример и исследуем несколько новых аспектов разработки и применения шаблонов. А именно, рассмотрим нетипизированные аргументы, или аргументы-выражения, и применение массива для управления семейством наследования.

Начнем с простого шаблона массива, который позволяет задавать размер массива. Один из приемов, который был использован в последней версии шаблона `Stack` – использование динамического массива внутри класса и аргумента в конструкторе для задания количества элементов. Другой подход состоит в применении аргумента шаблона для задания размера обычного массива, и как раз так поступает новый шаблон `array` в C++11. В листинге 14.17 приведена более скромная версия.

Листинг 14.17. arraytp.h

```
// arraytp.h -- шаблон массива
#ifndef ARRAYTP_H_
#define ARRAYTP_H_
#include <iostream>
#include <cstdlib>
template <class T, int n>
class ArrayTP
{
private:
    T ar[n];
public:
    ArrayTP() {};
    explicit ArrayTP(const T & v);
    virtual T & operator[](int i);
    virtual T operator[](int i) const;
};
template <class T, int n>
ArrayTP<T,n>::ArrayTP(const T & v)
{
    for (int i = 0; i < n; i++)
        ar[i] = v;
}
template <class T, int n>
T & ArrayTP<T,n>::operator[](int i)
{
    if (i < 0 || i >= n)
    {
        std::cerr << "Error in array limits: " << i // выход за пределы допустимого
               << " is out of range\n";                  // диапазона индекса в массиве
        std::exit(EXIT_FAILURE);
    }
    return ar[i];
}
```

```

template <class T, int n>
T ArrayTP<T,n>::operator[] (int i) const
{
    if (i < 0 || i >= n)
    {
        std::cerr << "Error in array limits: " << i // выход за пределы допустимого
              << " is out of range\n";                  // диапазона индекса в массиве
        std::exit(EXIT_FAILURE);
    }
    return ar[i];
}
#endif

```

Обратите внимание на заголовок шаблона в листинге 14.17:

```
template <class T, int n>
```

Ключевое слово `class` (или эквивалентное в этом контексте `typename`) объявляет `T` как параметр типа, или аргумент типа. Ключевое слово `int` объявляет, что `n` имеет тип `int`. Такой вид параметра – определяющий конкретный тип, а не обобщенное имя типа – называется *нетипизированным параметром*, или *параметром-выражением*. Предположим, что имеется следующее определение:

```
ArrayTP<double, 12> eggweights;
```

Встретив его, компилятор определит класс `ArrayTP<double, 12>` и создаст объект `eggweights` этого класса. При определении класса компилятор заменит `T` на `double` и `n` на `12`.

Аргументы-выражения имеют некоторые ограничения. Аргумент-выражение может быть целочисленного типа, перечислимого типа, ссылкой или указателем. Поэтому объявление `double m` является недопустимым, тогда как `double & rm` и `double * pm` допускаются. Кроме того, код шаблона не может изменять значение аргумента или использовать его адрес. Например, в шаблоне `ArrayTP` выражения `n++` или `&n` не разрешены. При инициализации шаблона значение, используемое для аргумента-выражения, должно быть константным выражением.

Такой способ установки размера массива обладает одним преимуществом перед вариантом с конструктором, применяемым в `Stack`. Вариант с конструктором использует память типа кучи, управляемую операциями `new` и `delete`, а вариант с аргументом-выражением – стек памяти для автоматических переменных. Второй способ быстрее, особенно в случае множества небольших массивов.

Главный недостаток подхода с аргументами-выражениями состоит в том, что для каждого размера массива генерируется собственный шаблон. Так, следующие объявления генерируют два отдельных определения классов:

```
ArrayTP<double, 12> eggweights;
ArrayTP<double, 13> donuts;
```

Однако объявления, показанные ниже, генерируют только одно определение класса, а информация о размере передается конструктору этого класса:

```
Stack<int> eggs(12);
Stack<int> dunkers(13);
```

Другое отличие заключается в том, что вариант с конструктором более гибок, поскольку размер массива хранится как член класса, а не жестко закодирован в определении. Поэтому можно, например, определить присваивание массива одного размера массиву другого размера или создать класс с массивами переменной размерности.

Универсальность шаблонов

В шаблонных классах можно применять те же приемы программирования, что и в обычных классах. Шаблонные классы могут выступать как в качестве базовых классов, так и компонентов других классов.

Например, можно создать шаблон стека на основе шаблона массива. Или можно взять шаблон массива и применить его для создания массива, элементы которого являются стеками, основанными на шаблоне стека. Это значит, что возможен такой код:

```
template <typename T> // или <class T>
class Array
{
private:
    T entry;
    ...
};

template <typename Type>
class GrowArray : public Array<Type> {...}; // наследование

template <typename Tp>
class Stack
{
    Array<Tp> ar; // использует Array<> в качестве компонента
    ...
};

...
Array < Stack<int> > asi; // массив стеков значений int
```

В последнем операторе во избежание путаницы с операцией `>>`, в C++98 требуется разделять два символа `>` как минимум одним пробельным символом. В C++11 это требование отсутствует.

Рекурсивное использование шаблонов

Другим примером универсальности шаблонов является возможность рекурсивного использования шаблонов. Например, приведенное ранее определение шаблона массива можно использовать так:

```
ArrayTP<ArrayTP<int, 5>, 10> twodee;
```

Здесь создается массив `twodee`, состоящий из 10 элементов, каждый из которых, в свою очередь, является массивом из пяти целых чисел (`int`). Эквивалентный обычный массив объявляется следующим образом:

```
int twodee[10][5];
```

Обратите внимание, что в синтаксисе шаблона размеры массива приведены в порядке, отличном от эквивалентного обычного двумерного массива. Эта идея проверяется в листинге 14.18. Также в нем для создания одномерного массива, содержащего суммы и средние значения для каждого из десяти наборов по пять чисел, применяется шаблон `ArrayTP`.

Вызов метода `cout.width(2)` приводит к выводу следующего элемента массива в виде двух символов (если для вывода целого числа не потребуется большая длина).

Листинг 14.18. twod.cpp

```
// twod.cpp -- создание двумерного массива
#include <iostream>
#include "arraytp.h"

int main(void)
{
    using std::cout;
    using std::endl;
    ArrayTP<int, 10> sums;
    ArrayTP<double, 10> aves;
    ArrayTP<ArrayTP<int, 5>, 10> twodee;
    int i, j;
    for (i = 0; i < 10; i++)
    {
        sums[i] = 0;
        for (j = 0; j < 5; j++)
        {
            twodee[i][j] = (i + 1) * (j + 1);
            sums[i] += twodee[i][j];
        }
        aves[i] = (double) sums[i] / 10;
    }
    for (i = 0; i < 10; i++)
    {
        for (j = 0; j < 5; j++)
        {
            cout.width(2);
            cout << twodee[i][j] << ' ';
        }
        cout << ": sum = ";
        cout.width(3);
        cout << sums[i] << ", average = " << aves[i] << endl;
    }
    cout << "Done.\n";
    return 0;
}
```

Вывод программы из листинга 14.18 содержит по одной строке для каждого из 10 элементов twodee, которые представляют собой массивы из пяти элементов:

```
1 2 3 4 5 : sum = 15, average = 1.5
2 4 6 8 10 : sum = 30, average = 3
3 6 9 12 15 : sum = 45, average = 4.5
4 8 12 16 20 : sum = 60, average = 6
5 10 15 20 25 : sum = 75, average = 7.5
6 12 18 24 30 : sum = 90, average = 9
7 14 21 28 35 : sum = 105, average = 10.5
8 16 24 32 40 : sum = 120, average = 12
9 18 27 36 45 : sum = 135, average = 13.5
10 20 30 40 50 : sum = 150, average = 15
Done.
```

Использование нескольких параметров типа

Допускается создание шаблонов с несколькими параметрами типа. Предположим, что требуется класс, содержащий два вида значений. Для этой цели можно создать шаблонный класс Pair. (Между прочим, STL содержит подобный шаблон, который

называется pair.) В листинге 14.19 приведен небольшой пример. В нем методы `first() const` и `second() const` выводят хранимые значения, а методы `first()` и `second()`, благодаря возврату ссылок на данные-члены класса `Pair`, позволяют присвоить хранимым величинам новые значения.

Листинг 14.19. pairs.cpp

```
// pairs.cpp -- определение и использование шаблона Pair
#include <iostream>
#include <string>
template <class T1, class T2>
class Pair
{
private:
    T1 a;
    T2 b;
public:
    T1 & first();
    T2 & second();
    T1 first() const { return a; }
    T2 second() const { return b; }
    Pair(const T1 & aval, const T2 & bval) : a(aval), b(bval) { }
    Pair() {}
};

template<class T1, class T2>
T1 & Pair<T1,T2>::first()
{
    return a;
}
template<class T1, class T2>
T2 & Pair<T1,T2>::second()
{
    return b;
}
int main()
{
    using std::cout;
    using std::endl;
    using std::string;
    Pair<string, int> ratings[4] =
    {
        Pair<string, int>("The Purpled Duck", 5),
        Pair<string, int>("Jaquie's Frisco Al Fresco", 4),
        Pair<string, int>("Cafe Souffle", 5),
        Pair<string, int>("Bertie's Eats", 3)
    };
    int joints = sizeof(ratings) / sizeof (Pair<string, int>);
    cout << "Rating:\t Eatery\n";           // вывод рейтингов закусочных
    for (int i = 0; i < joints; i++)
        cout << ratings[i].second() << ":\t "
            << ratings[i].first() << endl;
    cout << "Oops! Revised rating:\n";       // вывод пересмотренного рейтинга
    ratings[3].first() = "Bertie's Fab Eats";
    ratings[3].second() = 6;
    cout << ratings[3].second() << ":\t "
        << ratings[3].first() << endl;
    return 0;
}
```

Обратите внимание, что в листинге 14.19 в функции `main()` для вызова конструкторов и в качестве аргумента для `sizeof` необходимо выражение `Pair<string, int>` — поскольку именем класса является `Pair<string, int>`, а не `Pair`.

А `Pair<char *, double>` представляет собой имя совершенно другого класса. Вывод программы из листинга 14.19 имеет следующий вид:

```
Rating: Eatery
5:      The Purpled Duck
4:      Jaquie's Frisco Al Fresco
5:      Cafe Souffle
3:      Bertie's Eats
Oops! Revised rating:
6:      Bertie's Fab Eats
```

Параметры типа по умолчанию в шаблонах

Еще одно новое свойство шаблонных классов — возможность указания значений по умолчанию для параметров типа:

```
template <class T1, class T2 = int> class Topo {...};
```

В этом случае компилятор использует `int` в качестве типа `T2`, если значение для `T2` отсутствует:

```
Topo<double, double> m1;           // тип T1 — double, тип T2 — double
Topo<double> m2;                   // тип T1 — double, тип T2 — int
```

Это свойство часто используется в STL (см. главу 16), если типом по умолчанию является класс.

Хотя можно задать значения по умолчанию для типов параметров шаблонных классов, для параметров шаблонных функций это сделать нельзя. Тем не менее, значения по умолчанию нетипизированных параметров можно указывать как для шаблонных классов, так и для шаблонных функций.

Специализации шаблона

Для шаблонов классов, как и шаблонов функций, возможны неявные создания экземпляров, явные создания экземпляров и явные специализации, которые все вместе также называются *специализациями*. Шаблон описывает класс через обобщенный тип, а специализация — это объявление класса, сгенерированное для конкретного типа.

Неявное создание экземпляров

В примерах шаблонов, которые вы видели до сих пор в этой главе, используется *неявное создание экземпляров*. При этом объявление одного или более объектов задает нужный тип, и компилятор генерирует на основе общего шаблона специализированное определение класса:

```
ArrayTP<int, 100> stuff;           // неявное создание экземпляра
```

Компилятор не создает неявное создание экземпляра класса, пока не потребуется его объект:

```
ArrayTP<double, 30> * pt;          // указатель, пока еще объекты не нужны
pt = new ArrayTP<double, 30>;      // теперь объект нужен
```

Второй оператор заставляет компилятор сгенерировать определение класса, а также объект, созданный согласно этому определению.

Явное создание экземпляров

Компилятор обеспечивает явное создание экземпляра объявления класса, если класс объявлен с применением ключевого слова `template`, а также указан необходимый тип или типы. Объявление класса должно находиться в том же пространстве имен, что и определение шаблона. Например, следующая строка кода объявляет, что `ArrayTP<string, 100>` является классом:

```
template class ArrayTP<string, 100>; //генерирует класс ArrayTP<string, 100>
```

В этом случае компилятор генерирует определение класса, включая определения методов, даже если не создаются или упоминаются объекты класса. Как и в случае неявного создания экземпляров, руководством для генерирования специализации служит общий шаблон.

Явная специализация

Явная специализация – это определение конкретного типа (или типов), который должен использоваться вместо общего шаблона. Иногда необходимо изменить шаблон так, чтобы он вел себя по-разному при создании экземпляров для различных типов – в этом случае можно создать явную специализацию. Предположим, например, что определен шаблон класса, представляющий отсортированный массив, элементы которого сортируются непосредственно при занесении в массив:

```
template <class T>
class SortedArray
{
    ... // подробности не показаны
};
```

Предположим также, что для сравнения значений шаблон использует операцию `>`. Она хорошо работает для чисел, а также в случаях, когда `T` является типом класса, в котором определен метод `T::operator>()`. Однако такой способ не сработает, если `T` является строкой, представляемой с помощью типа `const char *`. Вообще говоря, шаблон будет работать, но строки окажутся отсортированными не по алфавиту, а по адресам. Поэтому требуется определение класса, где вместо операции `>` используется сравнение `strcmp()`. В этом случае можно указать явную специализацию шаблона – т.е. шаблон, определенный для одного конкретного типа, а не общего типа. Если запросу специализации удовлетворяет и специализированный шаблон, и общий шаблон, компилятор использует специализированный вариант.

Определение специализированного шаблона класса имеет вид:

```
template <> class ИмяКласса<имя-специализированного-типа> { ... };
```

Некоторые старые компиляторы могут распознавать только ранние формы без префикса `template <>`:

```
class ИмяКласса<имя-специализированного-типа> { ... };
```

Для создания шаблона `SortedArray`, специализированного для типа `const char *`, в современной нотации нужен примерно такой код:

```
template <> class SortedArray<const char *>
{
    ... // подробности не показаны
};
```

Здесь для сравнения значений массива код реализации должен использовать вместо операции `>` функцию `strcmp()`. Теперь запросы шаблона `SortedArray` для типа `const char *` будут применять специализированное определение вместо более общего определения шаблона:

```
SortedArray<int> scores;           // используется общее определение
SortedArray<const char *> dates; // используется специализированное определение
```

Частичная специализация

В C++ разрешена *частичная специализация*, которая частично ограничивает общность шаблона. Например, используя частичную специализацию, можно задать конкретный тип для одного из параметров типа:

```
// Общий шаблон
template <class T1, class T2> class Pair {...};

// Специализация, в которой для T2 указан тип int
template <class T1> class Pair<T1, int> {...};
```

Угловые скобки `<>`, следующие за ключевым словом `template`, объявляют параметры типов, которые пока еще не специализированы. Таким образом, второе объявление указывает для `T2` тип `int`, но оставляет параметр `T1` открытым. Обратите внимание, что указание всех типов приводит к пустым угловым скобкам и получению завершенной явной специализации:

```
// Специализация, в которой для T1 и T2 указан тип int
template <> class Pair<int, int> {...};
```

Если у компилятора есть выбор, он применяет наиболее специальный шаблон. Вот что произойдет для трех приведенных выше шаблонов:

```
Pair<double, double> p1; // используется общий шаблон Pair
Pair<double, int> p2;    // используется частичная специализация Pair<T1, int>
Pair<int, int> p3;       // используется явная специализация Pair<int, int>
```

Можно частично специализировать существующий шаблон, введя специальную версию для указателей:

```
template<class T>          // общая версия
class Feeb { ... };
template<class T*>          // частичная специализация с указателем
class Feeb { ... };          // измененный код
```

Если предоставить тип, который не является указателем, компилятор задействует общую версию, а если использовать указатель, компилятор выберет специализацию с указателем:

```
Feeb<char> fb1;           // используется общий шаблон Feeb (T - это char)
Feeb<char *> fb2;         // используется специализация Feeb T* (T - это char)
```

Без частичной специализации для второго объявления будет выбран общий шаблон, интерпретирующий `T` как тип `char *`. А при частичной специализации будет выбран специализированный шаблон, интерпретирующий `T` как тип `char`.

Частичная специализация позволяет задавать различные ограничения. Например, пусть имеются следующие объявления:

```
// Общий шаблон
template <class T1, class T2, class T3> class Trio{...};
// Специализация, когда для T3 указан T2
template <class T1, class T2> class Trio<T1, T2, T2> {...};
```

```
// Специализация, когда для T3 и T2 указан T1*
template <class T1> class Trio<T1, T1*, T1*> {...};

Для этих объявлений компилятор выберет следующие варианты:
Trio<int, short, char *> t1;           // используется общий шаблон
Trio<int, short> t2;                   // используется Trio<T1, T2, T2>
Trio<char, char *, char *> t3;         // используется Trio<T1, T1*, T1*>
```

Шаблоны-члены

Шаблоны могут быть членами структуры, класса или шаблонного класса. Эти свойства необходимы библиотеке STL для полного определения своей структуры. В листинге 14.20 приведен небольшой пример шаблонного класса с вложенными в виде членов шаблонным классом и шаблонной функцией.

Листинг 14.20. tempmemb.cpp

```
// tempmemb.cpp – шаблоны-члены
#include <iostream>
using std::cout;
using std::endl;
template <typename T>
class beta
{
private:
    template <typename V>                      // вложенный шаблонный класс-член
    class hold
    {
private:
    V val;
public:
    hold(V v = 0) : val(v) {}
    void show() const { cout << val << endl; }
    V Value() const { return val; }
};
hold<T> q;                                // шаблонный объект
hold<int> n;                               // шаблонный объект
public:
    beta( T t, int i ) : q(t), n(i) {}
    template<typename U>                      // шаблонный метод
    U blab(U u, T t) { return (n.Value() + q.Value()) * u / t; }
    void Show() const { q.show(); n.show(); }
};
int main()
{
    beta<double> guy(3.5, 3);
    cout << "T was set to double\n";      // T установлен в double
    guy.Show();
    cout << "V was set to T, which is double, then V was set to int\n";
        // V установлен в T, который double, затем V установлен в int
    cout << guy.blab(10, 2.3) << endl;
    cout << "U was set to int\n";          // U установлен в int
    cout << guy.blab(10.0, 2.3) << endl;
    cout << "U was set to double\n";       // U установлен в double
    cout << "Done\n";
    return 0;
}
```

Шаблон `hold` объявлен в закрытом разделе, поэтому он доступен только в пределах класса `beta`. Класс `beta` использует шаблон `hold` для определения двух членов данных:

```
hold<T> q;           // шаблонный объект
hold<int> n;         // шаблонный объект
```

`n` – объект `hold`, основанный на типе `int`, а член `q` – объект `hold`, основанный на типе `T` (параметр шаблона `beta`). Следующее объявление в функции `main()` присваивает `T` тип `double`, а `q` – тип `hold<double>`:

```
beta<double> guy(3.5, 3);
```

В методе `blah()` один тип (`U`) определен неявно, с помощью значения аргумента при вызове метода, а другой тип (`T`) определен типом создания экземпляра объекта. В данном примере объявление для `guy` назначает `T` тип `double`. Первый аргумент при вызове метода в следующем операторе назначает `U` тип `int`, соответствующий значению 10:

```
cout << guy.blah(10, 2.5) << endl;
```

Таким образом, хотя автоматическое преобразование типов из-за смешения типов приводит к вычислению `blah()` как `double`, возвращаемое значение, имеющее тип `U`, должно быть `int`. Поэтому оно усекается до 28, как показано в выводе программы:

```
T was set to double
3.5
3
V was set to T, which is double, then V was set to int
28
U was set to int
28.2609
U was set to double
Done
```

Если в вызове `guy.blah()` заменить 10 на 10.0, то тип `U` будет установлен в `double`, поэтому типом возврата будет `double`, о чем говорит наличие 28.2609 в выводе.

Как упоминалось ранее, тип второго параметра в определении объекта `guy` устанавливается в `double`. Но в отличие от первого параметра, тип второго параметра не задается вызовом функции. Например, показанный ниже оператор по-прежнему реализует `blah()` как `blah(int, double)`, и значение 3 будет преобразовано в тип `double` по обычным правилам соответствия прототипам функций:

```
cout << guy.blah(10, 3) << endl;
```

Можно объявить класс `hold` и метод `blah` в шаблоне `beta` и определить их за пределами этого шаблона. Правда, некоторые старые компиляторы вообще не воспринимают шаблоны-члены, а другие допускают их в том виде, который представлен в листинге 14.20, но не разрешают определять вне класса. Однако при наличии современного компилятора можно определить шаблонные методы за пределами шаблона `beta` следующим образом:

```
template <typename T>
class beta
{
private:
    template <typename V> // объявление
```

```

class hold;
hold<T> q;
hold<int> n;
public:
beta(T t, int i) : q(t), n(i) {}
template<typename U> // объявление
U blab(U u, T t);
void Show() const { q.show(); n.show(); }
};

// Определение члена
template <typename T>
template<typename V>
class beta<T>::hold
{
private:
V val;
public:
hold(V v = 0) : val(v) {}
void show() const { std::cout << val << std::endl; }
V Value() const { return val; }
};

// Определение члена
template <typename T>
template <typename U>
U beta<T>::blab(U u, T t)
{
return (n.Value() + q.Value()) * u / t;
}

```

Определения должны идентифицировать `T`, `V` и `U` как параметры шаблона. Из-за вложенности шаблонов необходимо использовать синтаксис

```

template <typename T>
template <typename V>

```

а не синтаксис

```
template<typename T, typename V>
```

`hold` и `blab` в определениях должны быть заданы как члены класса `beta<T>`, и для этого применяется операция разрешения контекста.

Шаблоны как параметры

Мы уже видели, что шаблоны могут иметь параметры типа, такие как `typename T`, и нетипизированные параметры вроде `int n`. Шаблоны также могут иметь параметры, которые сами являются шаблонами. Это еще одно дополнение, связанное с шаблонами, которое использовалось для реализации STL.

В листинге 14.21 показан пример, который начинается со следующих строк:

```

template <template <typename T> class Thing>
class Crab

```

Здесь `template <template <typename T> class Thing>` – параметр-шаблон, причем `template <typename T> class` – тип, а `Thing` – параметр. Что это означает? Предположим, имеется объявление

```
Crab<King> legs;
```

Чтобы оно работало, аргумент шаблона King должен быть шаблонным классом, а его объявление должно соответствовать объявлению параметра-шаблона Thing:

```
template <typename T>
class King {...};
```

Класс Crab в листинге 14.21 объявляет два объекта:

```
Thing<int> s1;
Thing<double> s2;
```

Предыдущее объявление для legs привело бы к подстановке King<int> вместо Thing<int> и King<double> вместо Thing<double>. Однако в листинге 14.21 приведено следующее объявление:

```
Crab<Stack> nebula;
```

Поэтому в данном случае Thing<int> реализуется как Stack<int>, а Thing<double> – как Stack<double>. В общем, параметр шаблона Thing заменяется любым шаблонным типом, используемым в качестве аргумента шаблона в объявлении объекта Crab.

Объявление класса Crab основано на трех предположениях о шаблонном классе, представленном параметром Thing. Этот класс должен содержать методы push() и pop(), а эти методы должны иметь определенный интерфейс. Класс Crab может использовать любой шаблонный класс, который соответствует типу Thing и содержит методы push() и pop(). В этой главе рассматривается один такой класс – шаблон Stack, определенный в stacktp.h. Этот класс и применяется в примере.

Листинг 14.21. tempparm.cpp

```
// tempparm.cpp -- шаблоны как параметры
#include <iostream>
#include "stacktp.h"
template <template <typename T> class Thing>
class Crab
{
private:
    Thing<int> s1;
    Thing<double> s2;
public:
    Crab() {};
    // Предполагается, что класс thing имеет члены push() и pop()
    bool push(int a, double x) { return s1.push(a) && s2.push(x); }
    bool pop(int & a, double & x) { return s1.pop(a) && s2.pop(x); }
};
int main()
{
    using std::cout;
    using std::cin;
    using std::endl;
    Crab<Stack> nebula;
    // Stack должен соответствовать шаблону template <typename T> class Thing
    int ni;
    double nb;
    cout << "Enter int double pairs, such as 4 3.5 (0 0 to end):\n";
    // Ввод пар чисел int и double
    while (cin >> ni >> nb && ni > 0 && nb > 0)
    {
        if (!nebula.push(ni, nb))
            break;
    }
}
```

```

while (nebula.pop(ni, nb))
    cout << ni << ", " << nb << endl;
cout << "Done.\n";
return 0;
}

```

Ниже показан пример запуска программы из листинга 14.21:

```

Enter int double pairs, such as 4 3.5 (0 0 to end) :
50 22.48
25 33.87
60 19.12
0 0
60, 19.12
25, 33.87
50, 22.48
Done.

```

Шаблонные параметры допускается смешивать с обычными параметрами. Например, объявление класса *Crab* может начинаться так:

```

template <template <typename T> class Thing, typename U, typename V>
class Crab
{
private:
    Thing<U> s1;
    Thing<V> s2;
    ...

```

Сейчас типы, сохраняемые в членах *s1* и *s2*, являются обобщенными, а не жестко закодированными типами. Поэтому в программе потребуется изменить определение *nebula* следующим образом:

```
Crab<Stack, int, double> nebula; // T=Stack, U=int, V=double
```

Шаблонный параметр *T* является шаблонным типом, а параметры типов *U* и *V* — нешаблонными типами.

Шаблонные классы и друзья

У объявлений шаблонных классов также могут быть друзья, которые принадлежат одной из трех перечисленных ниже категорий.

- Нешаблонные друзья.
- Связанные шаблонные друзья — тип друга определяется типом класса при создании его экземпляра.
- Не связанные шаблонные друзья — все специализации друга являются друзьями для всех специализаций класса.

Рассмотрим пример для каждого случая.

Нешаблонные дружественные функции для шаблонных классов

Объявим в шаблонном классе обычную функцию в качестве друга:

```

template <class T>
class HasFriend
{
public:
    friend void counts(); // дружественная для всех созданий экземпляров HasFriend
    ...
};

```

Здесь функция `counts()` объявляется дружественной для всех возможных созданий экземпляров шаблона. Например, она будет дружественной для класса `HasFriend<int>` и для класса `HasFriend<string>`.

Функция `counts()` не вызывается объектом (она является дружественной, а не функцией-членом) и она не принимает каких-либо объектных параметров. Каким же образом она обращается к объекту `HasFriend`? Существует несколько вариантов. Она может иметь доступ к глобальному объекту; она может иметь доступ к локальным объектам через глобальный указатель; она может создать собственные объекты; и она может иметь доступ к статическим членам-данным, расположенным отдельно от объекта.

Предположим, что для дружественной функции требуется создать аргумент типа шаблонного класса. Возможно ли, например, следующее объявление друга?

```
friend void report(HasFriend &); // возможно ли?
```

Ответ – невозможно. Дело в том, что объект `HasFriend` не может существовать. Существуют только конкретные специализации, такие как `HasFriend<short>`. Для создания аргумента типа шаблонного класса необходимо указать специализацию, например:

```
template <class T>
class HasFriend
{
    friend void report(HasFriend<T> &); // связанный друг шаблона
    ...
};
```

Чтобы понять, что здесь происходит, представьте себе специализацию, генерируемую при объявлении объекта конкретного типа:

```
HasFriend<int> hf;
```

Компилятор заменит параметр шаблона `T` на `int`, и объявление друга примет следующий вид:

```
class HasFriend<int>
{
    friend void report(HasFriend<int> &); // связанный друг шаблона
    ...
};
```

То есть функция `report()` с параметром `HasFriend<int>` становится дружественной для класса `HasFriend<int>`. Аналогично, функция `report()` с параметром `HasFriend<double>` будет перегруженной версией `report()`, которая является дружественной для класса `HasFriend<double>`.

Обратите внимание, что `report()` – нешаблонная функция: шаблоном является лишь ее параметр. Это означает, что для использования друзей необходимо определить явные специализации:

```
void report(HasFriend<short> &) { ... }; // явная специализация для short
void report(HasFriend<int> &) { ... }; // явная специализация для int
```

Эти моменты демонстрируются в листинге 14.22. В шаблоне `HasFriend` имеется статический член `ct`. Это означает, что любая конкретная специализация класса содержит собственный статический член. Метод `counts()`, являющийся другом для всех специализаций `HasFriend`, выводит значения `ct` из двух конкретных специализаций – `HasFriend<int>` и `HasFriend<double>`. В программе имеются также две функции `reports()`, каждая из которых является дружественной для одной конкретной специализации `HasFriend`.

Листинг 14.22. frnd2tmp.cpp

```

// frnd2tmp.cpp -- шаблонный класс с нешаблонными друзьями
#include <iostream>
using std::cout;
using std::endl;
template <typename T>
class HasFriend
{
private:
    T item;
    static int ct;
public:
    HasFriend(const T & i) : item(i) {ct++;}
    ~HasFriend() {ct--;}
    friend void counts();
    friend void reports(HasFriend<T> &); // template parameter
};

// Каждая специализация имеет собственный статический член данных
template <typename T>
int HasFriend<T>::ct = 0;

// Нешаблонный друг для всех классов HasFriend<T>
void counts()
{
    cout << "int count: " << HasFriend<int>::ct << "; ";
    cout << "double count: " << HasFriend<double>::ct << endl;
}

// Нешаблонный друг для класса HasFriend<int>
void reports(HasFriend<int> & hf)
{
    cout << "HasFriend<int>: " << hf.item << endl;
}

// Нешаблонный друг для класса HasFriend<double>
void reports(HasFriend<double> & hf)
{
    cout << "HasFriend<double>: " << hf.item << endl;
}

int main()
{
    cout << "No objects declared: "; // объекты пока не объявлены
    counts();

    HasFriend<int> hfi1(10);
    cout << "After hfi1 declared: "; // после объявления hfi1
    counts();

    HasFriend<int> hfi2(20);
    cout << "After hfi2 declared: "; // после объявления hfi2
    counts();

    HasFriend<double> hfdb(10.5);
    cout << "After hfdb declared: "; // после объявления hfdb
    counts();
    reports(hfi1);
    reports(hfi2);
    reports(hfdb);
    return 0;
}

```

Некоторые компиляторы могут выдать предупреждение об использовании нешаблонной дружественной функции. Ниже показан вывод программы из листинга 14.22:

```
No objects declared: int count: 0; double count: 0
After hfil declared: int count: 1; double count: 0
After hfi2 declared: int count: 2; double count: 0
After hfdb declared: int count: 2; double count: 1
HasFriend<int>: 10
HasFriend<int>: 20
HasFriend<double>: 10.5
```

Связанные шаблонные функции, дружественные шаблонным классам

Рассмотренный выше пример можно изменить, сделав дружественные функции также шаблонами. В частности, можно создать связанных друзей шаблона — чтобы каждая специализация класса получала соответствующую специализацию друга. Этот прием немного сложнее, чем в случае нешаблонных друзей, и состоит из трех шагов.

На первом шаге перед определением класса необходимо объявить каждую шаблонную функцию:

```
template <typename T> void counts();
template <typename T> void report(T &);
```

Затем внутри функции нужно снова объявить шаблоны в качестве друзей. Вот операторы, которые объявляют специализации, основанные на типе параметра шаблонного класса:

```
template <typename TT>
class HasFriendT
{
    ...
    friend void counts<TT>();
    friend void report<>(HasFriendT<TT> &);
};
```

Угловые скобки `<>` в объявлениях означают специализации шаблона. В случае `report()` скобки `<>` могут быть пустыми, т.к. аргумент типа шаблона можно получить из аргумента функции

```
HasFriendT<TT>
```

Но возможен и такой вариант:

```
report< HasFriendT<TT> >(HasFriendT<TT> &)
```

Функция `counts()` не имеет параметров, поэтому для определения ее специализации нужно задействовать аргумент шаблона (`<TT>`). Обратите внимание, что `TT` — тип параметра для класса `HasFriendT`. Чтобы понять эти объявления, лучше представить, чем они станут при объявлении объекта конкретной специализации. Предположим, например, что объявлен такой объект:

```
HasFriendT<int> squack;
```

Компилятор подставит вместо `TT` тип `int` и сгенерирует следующее определение класса:

```
class HasFriendT<int>
{
    ...
    friend void counts<int>();
    friend void report<>(HasFriendT<int> &);
};
```

Одна специализация основана на типе TT , который преобразуется в int , а другая — на $\text{HasFriendT}\langle\text{TT}\rangle$, который преобразуется в $\text{HasFriendT}\langle\text{int}\rangle$. Таким образом, специализации шаблона $\text{counts}\langle\text{int}\rangle()$ и $\text{report}\langle\text{HasFriendT}\langle\text{int}\rangle\rangle()$ объявлены как друзья класса $\text{HasFriendT}\langle\text{int}\rangle$.

Третье требование, которому должна удовлетворять программа — она должна содержать определения шаблонов для друзей. Все эти три аспекта демонстрируются в листинге 14.23. Обратите внимание, что в листинге 14.22 одна функция $\text{counts}()$ является другом для всех классов HasFriend . А в листинге 14.23 имеются две функции $\text{counts}()$, но лишь одна из них является другом каждому созданному типу класса. Поскольку вызовы функции $\text{counts}()$ не содержат параметров, из которых компилятор мог бы вывести требуемую специализацию, в этих вызовах используются формы $\text{count}\langle\text{int}\rangle()$ и $\text{count}\langle\text{double}\rangle()$. Но в вызовах $\text{reports}()$ компилятор может определять специализацию на основе типа аргумента. С тем же результатом можно использовать и форму $\langle\rangle$:

```
report<HasFriendT<int>>(hfi2); // эквивалентно report(hfi2);
```

Листинг 14.23. tmp2tmp.cpp

```
// tmp2tmp.cpp -- шаблонные друзья для шаблонного класса
#include <iostream>
using std::cout;
using std::endl;

// Прототипы шаблонов
template <typename T> void counts();
template <typename T> void report(T &);

// Шаблонный класс
template <typename TT>
class HasFriendT
{
private:
    TT item;
    static int ct;
public:
    HasFriendT(const TT & i) : item(i) {ct++;}
    ~HasFriendT() { ct--; }
    friend void counts<TT>();
    friend void report<>(HasFriendT<TT> &);
};

template <typename T>
int HasFriendT<T>::ct = 0;

// Определения дружественных функций для шаблона
template <typename T>
void counts()
{
    cout << "template size: " << sizeof(HasFriendT<T>) << "; "; // размер шаблона
    cout << "template counts(): " << HasFriendT<T>::ct << endl; // counts() из шаблона
}

template <typename T>
void report(T & hf)
{
    cout << hf.item << endl;
}
```

```

int main()
{
    counts<int>();
    HasFriendT<int> hfi1(10);
    HasFriendT<int> hfi2(20);
    HasFriendT<double> hfdb(10.5);
    report(hfi1); // генерирует report(HasFriendT<int> &)
    report(hfi2); // генерирует report(HasFriendT<int> &)
    report(hfdb); // генерирует report(HasFriendT<double> &)
    cout << "counts<int>() output:\n"; // вывод из counts<int>()
    counts<int>();
    cout << "counts<double>() output:\n"; // вывод из counts<double>()
    counts<double>();
    return 0;
}

```

Вывод программы из листинга 14.23 выглядит следующим образом:

```

template size: 4; template counts(): 0
10
20
10.5
counts<int>() output:
template size: 4; template counts(): 2
counts<double>() output:
template size: 8; template counts(): 1

```

Как видите, `counts<double>` сообщает размер шаблона, отличный от выводимого `counts<int>` – т.е. каждому типу `T` соответствует собственная дружественная функция `count()`.

Не связанные шаблонные функции, дружественные шаблонным классам

В предыдущем разделе связанные шаблонные дружественные функции являются специализациями для шаблона, определенного вне класса. Специализация `int` класса дает специализацию функции `int` и т.д. Объявив шаблон внутри класса, можно создать не связанные дружественные функции, когда любая специализация функции будет дружественной для любой специализации класса. У не связанных друзей параметры типа для шаблонов друзей отличаются от параметров типа для шаблонных классов:

```

template <typename T>
class ManyFriend
{
    ...
    template <typename C, typename D> friend void show2(C &, D &);
};

```

В листинге 14.24 приведен пример применения не связанных друзей. В этом примере вызов `show2(hfi1, hfi2)` соответствует следующей специализации:

```

void show2<ManyFriend<int> &, ManyFriend<int> &>
    (ManyFriend<int> & c, ManyFriend<int> & d);

```

Поскольку данная функция является другом для всех специализаций `ManyFriend`, она имеет доступ к членам `item` всех специализаций. Однако она использует доступ только к объектам `ManyFriend<int>`.

Аналогично, вызов `show2(hfd, hfi2)` соответствует такой специализации:

```

void show2<ManyFriend<double> &, ManyFriend<int> &>
    (ManyFriend<double> & c, ManyFriend<int> & d);

```

794 Глава 14

Эта функции также является другом для всех специализаций `ManyFriend` и использует доступ к члену `item` объекта `ManyFriend<int>`, а также к члену `item` объекта `ManyFriend<double>`.

Листинг 14.24. `manyfrnd.cpp`

```
// manyfrnd.cpp - не связанная шаблонная функция, дружественная шаблонному классу
#include <iostream>
using std::cout;
using std::endl;

template <typename T>
class ManyFriend
{
private:
    T item;
public:
    ManyFriend(const T & i) : item(i) {}
    template <typename C, typename D> friend void show2(C &, D &);
};

template <typename C, typename D> void show2(C & c, D & d)
{
    cout << c.item << ", " << d.item << endl;
}

int main()
{
    ManyFriend<int> hfil(10);
    ManyFriend<int> hfi2(20);
    ManyFriend<double> hfdb(10.5);
    cout << "hfil, hfi2: ";
    show2(hfil, hfi2);
    cout << "hfdb, hfi2: ";
    show2(hfdb, hfi2);
    return 0;
}
```

Ниже показан вывод программы из листинга 14.24:

```
hfil, hfi2: 10, 20
hfdb, hfi2: 10.5, 20
```

Псевдонимы шаблонов (C++11)

Бывает удобно, особенно при построении шаблонов, создавать псевдонимы для типов. Конструкция `typedef` позволяет создавать псевдонимы для специализаций шаблонов:

```
// Определение трех псевдонимов с помощью typedef
typedef std::array<double, 12> arrd;
typedef std::array<int, 12> arri;
typedef std::array<std::string, 12> arrst;
arrd gallons;           // gallons имеет тип std::array<double, 12>
arri days;              // days имеет тип std::array<int, 12>
arrst months;           // months имеет тип std::array<std::string, 12>
```

Но если приходится постоянно писать код, содержащий такие описания `typedef`, вы можете подумать, а не забыли ли вы какую-то языковую возможность, которая упрощает эту задачу, или не забыли ли добавить такую возможность в язык его разработ-

чики. В C++11, наконец, появилась ранее недоступная возможность — способ использовать шаблон для получения семейства псевдонимов. Вот как это выглядит:

```
template<typename T>
using arrtype = std::array<T, 12>; // шаблон для создания
// нескольких псевдонимов
```

Ниже объявлен псевдоним шаблона `arrtype`, который можно применять вместо спецификатора типа:

```
arrtype<double> gallons; // gallons имеет тип std::array<double, 12>
arrtype<int> days; // days имеет тип std::array<int, 12>
arrtype<std::string> months; // months имеет тип std::array<std::string, 12>
```

Короче говоря, `arrtype<T>` означает тип `std::array<T, 12>`.

В C++11 синтаксис `using` можно использовать и не для шаблонов. В таких случаях он эквивалентен `typedef`:

```
typedef const char * pc1; // синтаксис typedef
using pc2 = const char *; // синтаксис using =
typedef const int *(*pa1)[10]; // синтаксис typedef
using pa2 = const int *(*)[10]; // синтаксис using =
```

По мере привыкания, эта новая форма окажется более понятной, т.к. она более четко отделяет имя типа от информации об этом типе.

В C++11 появилось еще одно дополнение — *шаблон с переменным числом аргументов* (*variadic template*), который позволяет определить шаблонный класс или шаблонную функцию с переменным количеством инициализаторов. Эта тема рассматривается в главе 18.

Резюме

В C++ имеется несколько средств для повторного использования кода. Общедоступное наследование, рассмотренное в главе 13, позволяет моделировать отношение *является*, когда производные классы могут повторно использовать код базовых классов. Закрытое и защищенное наследование также позволяет повторно использовать код базовых классов, но в этом случае моделируется отношение *содержит*. При закрытом наследовании открытые и защищенные члены базового класса становятся закрытыми членами производного класса. При защищенном наследовании открытые и защищенные члены базового класса становятся защищенными членами производного класса. То есть в обоих случаях открытый интерфейс базового класса становится внутренним интерфейсом для производного класса. Иногда это называют наследованием реализации, а не интерфейса, т.к. производный объект не может явно использовать интерфейс базового класса. Поэтому производный объект нельзя считать разновидностью базового объекта. А из-за этого указатель или ссылку на базовый объект нельзя применять для ссылки на объект производного класса без явного приведения типа.

Еще один способ повторного использования кода класса — разработка класса, члены которого сами являются объектами. Этот подход называется *включением*, *иерархическим представлением* или *композицией* и также моделирует отношение *содержит*. Включение проще реализовать и применять, чем закрытое или защищенное наследование, и поэтому оно используется чаще. Однако возможности закрытого и защищенного наследования слегка различаются. Например, наследование позволяет производному классу обращаться к защищенным членам базового класса. Оно также позволяет производному классу переопределять виртуальные функции, унаследованные от базо-

вого класса. Включение не является разновидностью наследования и поэтому не обеспечивает таких возможностей. Зато включение удобнее, если нужно создать несколько объектов одного класса. Например, класс *Country* (Страна) может содержать массив объектов *State* (Штат).

Множественное наследование позволяет использовать в классе код нескольких других классов. Закрытое и защищенное множественное наследование приводит к созданию отношений *содержит*, а открытое множественное наследование — к созданию отношений *является*. Применение множественного наследования приводит к возникновению проблем, связанных с неоднозначностью имен и неоднозначным наследованием базового класса. Для разрешения неоднозначности имен можно использовать квалификаторы класса, а для преодоления неоднозначности наследования — виртуальные базовые классы. Однако виртуальные базовые классы вводят новые правила для списка инициализации в конструкторах и для разрешения неоднозначности.

Шаблоны классов позволяют создать общую структуру класса, в которой тип (как правило, тип члена) представлен параметром типа. Обычно шаблон выглядит следующим образом:

```
template <class T>
class Ic
{
    T v;
    ...
public:
    Ic(const T & val) : v(val) { }
    ...
};
```

Здесь *T* — параметр типа, и он играет роль заполнителя для реального типа, который будет указан позднее. (Этот параметр может иметь любое допустимое в C++ имя, но обычно применяется *T* или *Type*.) В данном контексте слово *class* можно заменить словом *typename*:

```
template <typename T> // эквивалентно template <class T>
class Rev {...} ;
```

Определение класса (создание экземпляра) генерируется при объявлении объекта класса и указании конкретного типа. Например, следующее объявление указывает компилятору сгенерировать объявление класса, в котором каждое вхождение параметра типа *T* в шаблоне заменено типом *short*:

```
class Ic<short> sic; // неявное создание экземпляра
```

В этом случае именем класса будет *Ic<short>*, а не *Ic*. Объявление *Ic<short>* называется *специализацией* шаблона. В данном случае это неявное создание экземпляра.

Явное создание экземпляра происходит при объявлении конкретной специализации класса с помощью ключевого слова *template*:

```
template class IC<int>; // явное создание экземпляра
```

В этом случае компилятор использует общий шаблон для генерации специализации *Ic<int>*, даже если еще не затребован ни один объект этого класса.

Можно создать явную специализацию — специализированное определение класса, которое переопределяет определение шаблона. Для этого определение класса начинается с конструкции *template<>*, потом указывается имя шаблонного класса, а за ним — угловые скобки, содержащие тип требуемой специализации.

Например, можно создать специализированный класс *Ic* для указателей на символы:

```
template <> class Ic<char *>.
{
    char * str;
    ...
public:
    Ic(const char * s) : str(s) { }
    ...
};
```

Тогда объявление следующего вида будет использовать не общий шаблон, а специализированное определение для `chic`:

```
class Ic<char *> chic;
```

Шаблон класса может задавать несколько общих типов и может иметь нестилизированные параметры:

```
template <class T, class TT, int n>
class Pals {...};
```

Показанное ниже объявление сгенерирует неявное создание экземпляра, заменив `T` на `double`, `TT` на `string` и `n` на `6`:

```
Pals<double, string, 6> mix;
```

Шаблон класса может иметь параметры, которые сами являются шаблонами:

```
template <template <typename T> class CL, typename U, int z>
class Trophy {...};
```

Здесь `z` – это значение типа `int`, `U` – имя типа, а `CL` – шаблон класса, определенного конструкцией `template <typename T>`.

Шаблоны класса могут быть специализированными частично:

```
template <class T> Pals<T, T, 10> {...};
template <class T, class TT> Pals<T, TT, 100> {...};
template <class T, int n> Pals <T, T*, n> {...};
```

В первом примере создается специализация, в которой оба типа одинаковы, а `n` имеет значение 6. Во втором примере создается специализация для `n`, равного 100. В третьем примере создается специализация, в которой второй тип является указателем на первый тип.

Шаблонные классы могут быть членами других классов, структур и шаблонов.

Главная цель создания всех рассмотренных методов – предоставить программисту возможность повторного использования проверенного кода без его ручного копирования. Это упрощает программирование и повышает надежность программ.

Вопросы для самоконтроля

- Для каждого набора классов из столбца А укажите, какое наследование – общедоступное или закрытое – лучше подходит для столбца Б.

А	Б
class Bear (Медведь)	class PolarBear (Белый медведь)
class Kitchen (Кухня)	class Home (Дом)
class Person (Человек)	class Programmer (Программист)
class Person (Человек)	class HorseAndJockey (Лошадь и жокей)
class Person, class Automobile (Человек, Автомобиль)	class Drive (Поездка)