

КАРСКИНЪ COMPUTER SCIENCE

ОБЪЕКТНО-ОРИЕНТИРОВАННОЕ ПРОГРАММИРОВАНИЕ

В C++

4-Е ИЗДАНИЕ



Р. ЛАФОРЕ

SAMS

ПИТЕР®

Глава 14

Шаблоны и исключения

- ◆ Шаблоны функций
- ◆ Шаблоны классов
- ◆ Исключения

В этой главе рассматриваются два важных инструмента C++ — шаблоны и исключения. Шаблоны позволяют использовать одни и те же функции или классы для обработки данных разных типов. Исключения дают возможность обрабатывать ошибки, возникающие в классах, удобным и универсальным способом. Эти две темы объединены в одной главе скорее по историческим причинам: эти два инструмента были введены в состав языка C++ практически одновременно. Они не входили в исходную спецификацию языка, но были представлены в виде «экспериментальных» в книге Эллиса и Страуструпа (Ellis and Stroustrup, 1990, см. приложение 3 «Библиография»), но постепенно они стали частью стандартного C++.

Концепция шаблонов может быть использована в двух видах: по отношению к функциям и по отношению к классам. Сначала мы взглянем на шаблоны функций, затем на шаблоны классов, а закончим главу разговором о механизме исключений.

Шаблоны функций

Допустим, вам нужно написать функцию вычисления модуля чисел. Из школьного курса алгебры вы, конечно, знаете, что модуль — это абсолютное значение числа, то есть число без знака. Напоминаем: модуль числа 3 равен 3, модуль числа -3 равен 3.

Скорее всего, функция вычисления модуля будет использоваться с каким-либо одним типом данных:

```
int abs(int n)           //вычисление модулей целых чисел
{
    return (n<0) ? -n : n; //если отрицательное, вернуть -n
}
```

Определенная выше функция берет аргумент типа `int` и возвращает результат того же типа. Но теперь представьте, что понадобилось найти модуль числа типа `long`. Придется писать еще одну функцию:

```
long abs(long n)           //вычисление модулей чисел типа long
{
    return (n<0) ? -n : n;
}
```

Да, теперь еще одну для float:

```
float abs(float n)        //модуль целых чисел
{
    return (n<0) ? -n : n; //если отрицательное, вернуть -n
}
```

Тело функций, как видите, ничем не отличается. И все же они совершенно разные, поскольку обрабатывают аргументы и возвращают значения разных типов. Они могут быть перегружены и иметь одинаковые имена, это правда, но все равно для каждой из них нужно писать отдельное определение. (Что касается языка С, в котором нет понятия перегрузки, нельзя использовать даже одинаковые имена для обозначения функций разных типов. Там это приведет к созданию целого клана функций с похожими названиями, таких, как `abs()`, `fabs()`, `fabsl()`, `labs()` и `cabs()`.)

Многократное переписывание этих функций-близнецов очень утомляет, затрудняет читабельность листинга, зато делает его увесистым. Можно показать неопытному заказчику тома исходных кодов, демонстрируя свою напряженную работу над проектом. А если где-то в алгоритме попадется ошибка, придется исправлять ее в теле каждой функции. Причем некорректное исправление ошибки приведет к несостоятельности программы.

Было бы здорово, если бы был какой-то способ написать эту функцию всего один раз и заставить ее работать с различными типами данных, возвращая, соответственно, результаты разного типа. Это как раз то, для чего существуют шаблоны функций в C++. Эта простая и гениальная идея схематично изображена на рис. 14.1.

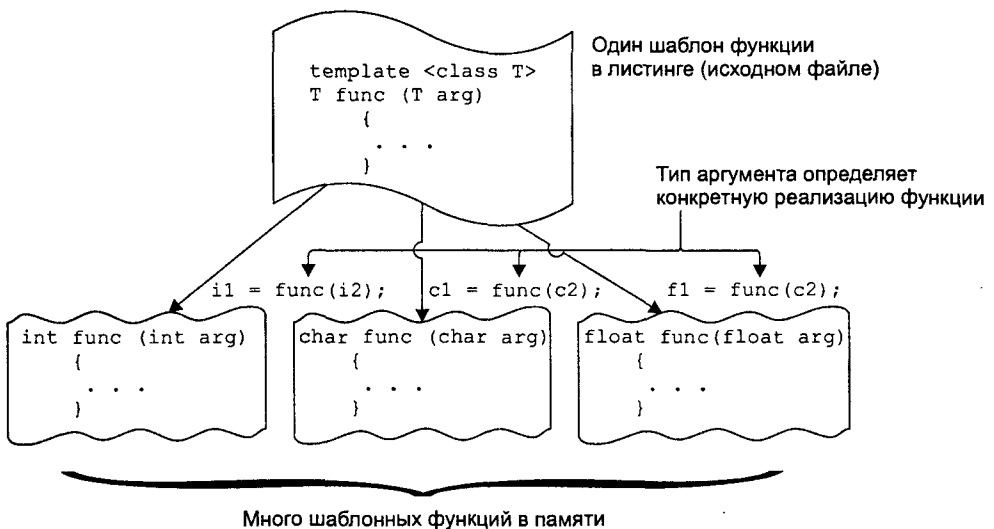


Рис. 14.1. Шаблон функции

Шаблон простой функции

Первый пример в этой главе показывает, как пишется шаблон функции, вычисляющей модуль числа. Шаблон работает со всеми базовыми числовыми типами. В программе сначала определяется шаблонная версия `abs()`, а затем в `main()` производится проверка правильности ее работы.

Листинг 14.1. Программа TEMPABS¹

```
// tempabs.cpp
// Шаблон функции вычисления модуля числа
#include <iostream>
using namespace std;
//-----
template <class T>          //Шаблон функции!
T abs(T n)
{
    return (n < 0) ? -n : n;
}
//-----
int main()
{
    int int1 = 5;
    int int2 = -6;
    long lon1 = 70000L;
    long lon2 = -80000L;
    double dub1 = 9.95;
    double dub2 = -10.15;

    //осуществления вызовов
    cout << "\nabs(" << int1 << ")=" << abs(int1); //abs(int)
    cout << "\nabs(" << int2 << ")=" << abs(int2); //abs(int)
    cout << "\nabs(" << lon1 << ")=" << abs(lon1); //abs(long)
    cout << "\nabs(" << lon2 << ")=" << abs(lon2); //abs(long)
    cout << "\nabs(" << dub1 << ")=" << abs(dub1);
                                //abs(double)
    cout << "\nabs(" << dub2 << ")=" << abs(dub2);
                                //abs(double)

    cout << endl;
    return 0;
}
```

Результаты работы программы:

```
abs(5)=5
abs(-6)=6
abs(70000)=70000
abs(-80000)=80000
abs(9.95)=9.95
abs(-10.15)=10.15
```

Итак, теперь функция `abs()` может работать со всеми тремя типами данных (`int`, `long`, `double`). Типы определяются функцией при передаче аргумента. Она будет работать и с другими базовыми числовыми типами, да даже и с пользовательскими, лишь бы был надлежащим образом перегружен оператор «меньше» (`<`) и унарный оператор «-».

¹ Все исходные тексты, приведенные в книге, вы найдете по адресу <http://www.piter.com/download>.

Вот спецификация функции `abs()` для работы с несколькими типами данных:

```
template <class T>           //шаблон функции
T abs (T n)
{
    return (n<0) ? -n : n;
}
```

Вся конструкция, начиная от ключевого слова `template` в начале первой строки и включая определение функции, называется *шаблоном функции*. Каким образом достигается такая потрясающая гибкость использования функции?

Синтаксис шаблона функции

Краеугольным камнем концепции шаблонов функции является представление использующегося функцией типа не в виде какого-либо специфического (например, `int`), а с помощью названия, вместо которого может быть подставлен *любой* тип. В приведенном примере таким именем является `T` (никакого сакрального смысла именно в таком названии нет. На его месте может все что угодно, например `anyType` или `YooHoo`). Ключевое слово `template` сообщает компилятору о том, что «сейчас будет море крови»: мы определяем шаблон функции. Ключевое слово `class`, заключенное в угловые скобки, можно с тем же успехом заменить на `type`. Как вы уже видели, можно определять собственные типы, используя классы, поэтому разницы между типами и классами в известном смысле нет совсем. Переменная, следующая за словом `class` (`T` в нашем примере), называется *аргументом шаблона*. Внутри всего определения шаблона любой конкретный тип данных, такой, как `int`, заменяет аргумент `T`. В `abs()` это имя встречается лишь дважды в первой строке в качестве типа аргумента и одновременно в качестве типа функции. В более сложном случае оно могло бы встретиться много раз, в том числе в теле функции.

Действия компилятора

Что компилятор делает, увидев ключевое слово `template` и следующее за ним определение функции? Правильно, почти ничего или очень мало. Дело в том, что сам по себе шаблон не вызывает генерацию компилятором какого-либо кода. Да и не может он этого сделать, поскольку еще не знает, с каким типом данных будет работать данная функция. Он разве что запоминает шаблон для будущего использования.

Генерации кода не происходит до тех пор, пока функция не будет реально вызвана в ходе выполнения программы. В примере `TEMPABS` это происходит в конструкциях типа `abs(int)` в выражении

```
cout << "\nabs(" << int1 << ")=" << abs(int1);
```

Когда компилятор видит такой вызов функции, он знает, что нужно использовать тип `int`, поскольку это тип аргумента `int1`, переданного в шаблон. Поэтому он генерирует код `abs` для типа `int`, подставляя его везде вместо `T`. Это называется *реализацией* шаблона функции; при этом каждый реализованный шаблон функции называется *шаблонной функцией*¹.

¹ То есть *шаблонная функция* — это реализация *шаблона функции*. О, великий, могучий, правдивый и свободный русский язык! — *Примеч. перев.*

Компилятор также генерирует вызов реализованной функции и вставляет его туда, где находится `abs(int1)`. Выражение `abs(lon1)` приводит к тому, что компилятором генерируется версия `abs(int)`, работающая с типом `long`, и ее вызов. То же самое касается типа `float`. Конечно, компилятор достаточно умен для того, чтобы создавать для каждого типа данных только одну копию `abs()`. Таким образом, несмотря на два вызова `int`-версии функции, в исполняемом коде она будет присутствовать только один раз.

Упрощение листинга

Имейте в виду, что объем оперативной памяти, занимаемой программой, не менялся из-за того, что мы стали использовать шаблон. Как было три одинаковых функций, так и осталось. Шаблоны спасают нас только от необходимости вручную писать три раза одно и то же в исходном файле. Однако это делает листинг короче. К тому же, если возникла задача изменения алгоритма, мы изменяем только одно тело функции вместо трех.

Решающий аргумент

Компилятор принимает решение о том, как именно компилировать функцию, основываясь только на типе данных используемого в шаблоне аргумента (или аргументов). Тип данных, возвращаемый функцией, не играет при этом роли. Это немного похоже на то, как компилятор принимает решение, какую из перегруженных функций вызывать.

Просто еще один трафарет

Мы уже поняли, что шаблон функции — это на самом деле не функция, так как она не приводит к созданию программного кода в памяти. Это просто модель, трафарет для создания множества функций из одной. Это вполне гармонирует с философией ООП. Как и шаблон, класс не представляет собой ничего конкретного, это только база для создания множества похожих объектов.

Шаблоны функций с несколькими аргументами

Взглянем на другой пример шаблона функции. У него три аргумента, два из которых шаблонные, а один — базового типа. Функция предназначена для поиска в массиве заданного числа. Она возвращает индекс найденного значения или `-1` в случае его отсутствия в массиве. Аргументами являются указатель на массив, значение, которое нужно искать, а также размер массива. В `main()` мы определяем четыре различных массива различных типов и четыре значения, которые нужно найти. Тип `char` в данном примере воспринимается как число. Для каждого массива вызывается шаблонная функция.

Листинг 14.2. Программа TEMPFIND

```
// tempfind.cpp
// Шаблон функции поиска в массиве
#include <iostream>
using namespace std;
```

```
//-----
// функция возвращает индекс или -1 при отсутствии //совпадения
template <class atype>
int find(atype* array, atype value, int size)
{
    for(int j=0; j<size; j++)
        if(array[j]==value)
            return j;
    return -1;
}
//-----
char chrArr[] = {1, 3, 5, 9, 11, 13}; //массив
char ch = 5; //искомое значение
int intArr[] = {1, 3, 5, 9, 11, 13};
int in = 6;
long lonArr[] = {1L, 3L, 5L, 9L, 11L, 13L};
long lo = 11L;
double dubArr[] = {1.0, 3.0, 5.0, 9.0, 11.0, 13.0};
double db = 4.0;

int main()
{
    cout << "\n 5 в chrArray: индекс=" << find(chrArr, ch, 6);
    cout << "\n 6 в intArray: индекс=" << find(intArr, in, 6);
    cout << "\n11 в lonArray: индекс=" << find(lonArr, lo, 6);
    cout << "\n 4 в dubArray: индекс=" << find(dubArr, db, 6);
    cout << endl;
    return 0;
}
```

Здесь, как видите, мы называем шаблонный аргумент именем `atype`. Оно появляется два раза в аргументах: как тип указателя на массив и как тип искомого значения. Третий аргумент — размер массива — всегда типа `int`. Это не шаблонный аргумент. Вот результаты работы программы:

```
5 в chrArray: индекс=2
6 в intArray: индекс=-1
11 в lonArray: индекс=4
4 в dubArray: индекс=-1
```

Компилятор генерирует четыре разных варианта функции, по одному на каждый тип. Функция находит число 5 в ячейке массива с индексом 2, не находит число 6 в массиве вообще и т. д.

Аргументы шаблона должны быть согласованы

При вызове шаблонной функции все экземпляры данного аргумента шаблона должны быть строго одного типа. Например, в `find()` есть массив типа `int`, соответственно, искомое значение должно быть того же типа. *Нельзя* делать так:

```
int intarray[]={1,3,5,7}; //массив int
float f1 = 5.0; //значение float
int value = find(intarray, f1, 4); //ошибка
```

Компилятору нужно, чтобы все экземпляры `atype` были одного типа. Он может сгенерировать код функции

```
find(int*, int, int):
```

но не станет заниматься такими глупостями:

```
find(int*, float, int);
```

потому что первый и второй аргумент должны быть одного типа.

Варианты синтаксиса

Некоторым программистам нравится все писать в строчку — и ключевое слово `template`, и объявление функции:

```
template<class type> int find(atype* array, atype value, int size)
{
    //набросаем тут тело функции
}
```

Конечно, добродушный компилятор счастлив и от такого оформления, но структуру синтаксиса лучше соблюдать, а надписи в одну строчку оставить для расписывания заборов. Все-таки должна быть определенная культура программирования.

Различные аргументы одного шаблона

В шаблоне функции можно использовать несколько шаблонных аргументов. Например, вы подумали, что неплохо бы написать шаблон функции `find()`, но понятия не имеете, к массивам какого размера его потом нужно будет применять. Если массив слишком большой, может понадобится тип `long` для обозначения его размера. С другой стороны, как-то не хочется по умолчанию применять `long`, потому что он может и не пригодиться. Хочется выбрать как тип размера массива, как и тип хранимых в нем данных прямо во время вызова функции. Для этого можно сделать размер массива еще одним аргументом шаблона. Назовем его `btype`:

```
template <class atype, class btype>
btype find(atype* array, atype value, btype size)
{
    for(btype j=0; j<size; j++) //использование btype.
        if(array[j]==value)
            return j;
    return static_cast<btype>(-1);
}
```

Теперь можно использовать значение как типа `int`, так и `long` (и даже любого другого пользовательского) в качестве размера массива. Компилятор при своей работе теперь будет ориентироваться не только на разные типы самого массива и искомого числа, но и на разные типы значений его размера.

Обратите внимание, что количество экземпляров шаблонной функции в памяти резко возрастает с ростом числа аргументов шаблона. Уже два разных аргумента в сочетании с шестью базовыми типами приводят к созданию в памяти 36 копий функции. Функции бывают довольно большими, и могут возникнуть проблемы с тем, что программе требуется слишком много памяти. С другой стороны, не всегда реально вызываются шаблонные функции, несмотря на наличие их шаблонов.

А почему не макросы?

Старосветские программисты, пишущие на С, снова удивятся, почему же мы не используем макросы для создания функций, работающих с разными типами данных? Например, функция `abs()` могла бы быть определена так:

```
#define abs(n) ( (n<0) ? (-n) : (n) )
```

Это возымело бы тот же эффект, что и от шаблона класса в `TEMPABS`, потому что здесь используется простое замещение одного текста на другой, а значит, можно указывать любые типы данных. Тем не менее, как уже отмечалось ранее, макросы редко используются в `C++`. С ними возникают некоторые проблемы. Одна из них заключается в том, что макросы не осуществляют проверку типов. Может быть несколько переменных макроса, которые обязаны быть одного типа, но компилятор не проверяет это условие. Еще одна проблема состоит в том, что не указан тип возвращаемого значения, поэтому компилятор, опять же, не может проверить корректность присваивания. В любом случае, макросы ограничены функциями, которые можно выразить с помощью всего одного выражения. Да есть еще и другие проблемы с макросами, которые привели к выходу их из моды при программировании на `C++`. В целом, лучше избегать их использования.

Что работает, а что нет

Как узнать, можно ли реализовать данную шаблонную функцию с тем или иным типом данных? Например, можно ли использовать функцию `find()` из программы `TEMPFIND` для нахождения `С`-строки (типа `char*`) в массиве таких строк? Чтобы выяснить этот факт, необходимо проверить операторы, используемые в функции. Если они все способны обработать такой тип данных, то, может быть, что-нибудь и получится. В `find()`, между тем, мы сравниваем две переменные с помощью оператора `==`, а его нельзя использовать с такими строками. Но можно обратиться к библиотечной функции `strcmp()`. Таким образом, работать с `С`-строками у нас не получилось. Зато наш шаблон функции работает с классом `string`, потому что в нем перегружен оператор сравнения (`==`).

Начните с обычной функции

При написании шаблонных функций, наверное, лучше начать с обычной функции, которая работает с фиксированным типом данных. Можно модифицировать и отлаживать ее сколько душе угодно, совершенно не беспокоясь о синтаксисе шаблонов и разнообразии типов. Затем, когда все доведено до блеска и работает без перебоев, можно вернуть определение функции в шаблон и проверить, как он будет работать с различными типами.

Шаблоны классов

Шаблонный принцип можно расширить и на классы. В этом случае шаблоны обычно используются, когда класс является хранилищем данных (хороший тому пример мы увидим в следующей главе «Стандартная библиотека шаблонов (STL)»).

Стеки и связные списки, рассмотренные нами в предыдущих главах, — тоже типичные примеры классов-хранилищ данных. До сих пор все приводимые примеры хранилищ могли содержать данные только одного базового типа. Класс `stack` из программы `STAKARAY` (глава 7 «Массивы и строки»), например, мог хранить только значения типа `int`. Вот немного сокращенная версия этого класса:

```
class Stack
{
private:
    int st[MAX];           //целочисленный массив
    int top;               //Индекс вершины стека
public:
    Stack():               //конструктор
    void push(int var);    //int-аргумент
    int pop();             //возвращает значение int
};
```

Если бы мы захотели хранить значения типа `long`, пришлось бы написать абсолютно новое определение класса:

```
class LongStack
{
private:
    long st[MAX];          //массив long
    int top;               //Индекс вершины стека
public:
    LongStack():           //конструктор
    void push(long var);   //int-аргумент
    long pop();            //возвращает значение int
};
```

Подобным же образом нам пришлось бы создавать классы для хранения данных каждого типа, если бы не шаблоны классов, которые и тут приходят к нам на помощь. С их помощью можно написать такую спецификацию класса, которая сохранит все, что попросят, без лишних вопросов. Создадим вариацию на тему `STAKARAY` для демонстрации возможностей шаблонов классов.

Листинг 14.3. Программа `TEMPSTAK`

```
// tempstak.cpp
// реализация стека в виде шаблона
#include <iostream>
using namespace std;
const int MAX = 100;
////////////////////////////////////
template <class Type>
class Stack
{
private:
    Type st[MAX];          //стек: массив любого типа
    int top;               //индекс вершины стека
public:
    Stack()                //конструктор
    { top = -1; }
    void push(Type var)    //занести число в стек
    { st[++top] = var; }
```

```

    Type pop()           //вынуть число из стека
    { return st[top--]; }
};
////////////////////////////////////
int main()
{
    Stack<float> s1;      //s1 - объект класса Stack<float>

    s1.push(1111.1F);    //занести 3 значения float,
    s1.push(2222.2F);    // вытолкнуть 3 значения float
    s1.push(3333.3F);
    cout << "1: " << s1.pop() << endl;
    cout << "2: " << s1.pop() << endl;
    cout << "3: " << s1.pop() << endl;

    Stack<long> s2;      //s2 - объект класса Stack<long>

    s2.push(123123123L); //занести 3 long, вытолкнуть 3 long
    s2.push(234234234L);
    s2.push(345345345L);
    cout << "1: " << s2.pop() << endl;
    cout << "2: " << s2.pop() << endl;
    cout << "3: " << s2.pop() << endl;
    return 0;
}

```

Здесь `Stack` является шаблонным классом. Идея шаблонных классов во многом сходна с идеей шаблонных функций. Ключевые слова `template` и `class Stack` говорят о том, что весь класс будет шаблонным.

```

template <class Type>
class Stack
{
    //Данные и методы используют шаблонный аргумент Type
};

```

Шаблонный аргумент `Type` используется вместо фиксированного типа во всех местах спецификации класса, где есть ссылка на тип массива `st`. Есть три таких места: определение `st`, тип аргумента функции `push()` и тип данных, возвращаемых функцией `pop()`.

Шаблоны классов отличаются от шаблонов функций способом реализации. Для создания шаблонной функции мы вызываем ее с аргументами нужного типа. Классы же реализуются с помощью определения объекта, использующего шаблонный аргумент:

```
Stack<float> s1;
```

Такое выражение создаст объект `s1`. Это стек, в котором хранятся числа типа `float`. Компилятор резервирует область памяти для данных этого объекта, используя тип `float` везде, где появляется аргумент `Type` в спецификации класса. Резервируется место и под методы класса (если они не были помещены в память другим объектом типа `Stack<float>`). Конечно же, методы используют только тип `float`. На рис. 14.2 показано, как шаблоны классов и определения конкретных объектов приводят к занесению этих объектов в память.

Создание объекта типа `Stack`, хранящего объекты каких-то других типов, как в выражении

```
Stack<long> s2;
```

означает не только резервирование другого объема памяти для данных, но и создание нового набора методов, оперирующих типом `long`.

Имейте в виду, что имя типа `s1` состоит из имени класса `Stack` и *шаблонного аргумента*: `Stack<float>`. Это отличает его от других классов, которые могут быть созданы из того же шаблона (`Stack<int>` или `Stack<long>`).

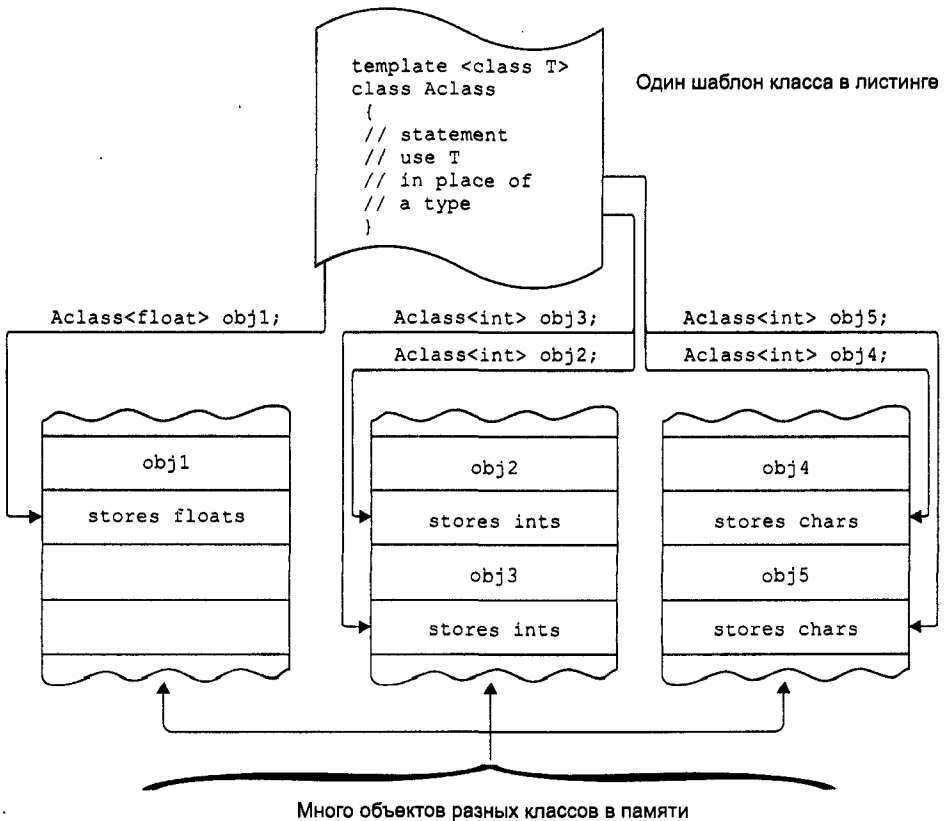


Рис. 14.2. Шаблон класса

В `TEMPSTAK` мы проверяем стеки `s1` и `s2`, внося в них и вытаскивая по три значения и выводя на экран все вытаскиваемые значения. Вот результаты работы программы:

```
1: 3333.3//стек float
2: 2222.2
3: 1111.1
1: 345345345//стек long
2: 234234234
3: 123123123
```

В этом примере мы «получаем два класса по цене одного». Для других типов данных можно создать свое множество объектов, и делается это всего одной строкой программы.

Контекстозависимое имя класса

В предыдущем примере все методы шаблона класса были определены внутри класса. Если методы определяются вне спецификации класса, нужно использовать новый синтаксис. Следующая программа показывает, как это можно сделать.

Листинг 14.4. Реализация стека в виде шаблона класса

```
// temstak2.cpp
// реализует стековый класс в виде шаблона
// методы определяются вне класса
#include <iostream>
using namespace std;
const int MAX = 100;
////////////////////////////////////
template <class Type>
class Stack
{
private:
    Type st[MAX];           //стек - массив любого типа
    int top;                //индекс вершины стека
public:
    Stack();                //конструктор
    void push(Type var);    //занести число в стек
    Type pop();             //взять число из стека
};
////////////////////////////////////
template<class Type>
Stack<Type>::Stack()        //конструктор
{
    top = -1;
}
//-----
template<class Type>
void Stack<Type>::push(Type var) //положить число в стек
{
    st[++top] = var;
}
//-----
template<class Type>
Type Stack<Type>::pop()      //взять число из стека
{
    return st[top--];
}
//-----

int main()
{
    Stack<float> s1;          //s1 - объект класса Stack<float>

    s1.push(1111.1F);        //занести 3 float. вытолкнуть 3 float
```

Листинг 14.4 (продолжение)

```

s1.push(2222.2F);
s1.push(3333.3F);
cout << "1: " << s1.pop() << endl;
cout << "2: " << s1.pop() << endl;
cout << "3: " << s1.pop() << endl;

Stack<long> s2;           //s2 - объект класса Stack<long>

s2.push(123123123L);     //занести 3 long. вытолкнуть 3 long
s2.push(234234234L);
s2.push(345345345L);
cout << "1: " << s2.pop() << endl;
cout << "2: " << s2.pop() << endl;
cout << "3: " << s2.pop() << endl;
return 0;
}

```

Выражение `template<class Type>` должно предварять не только определение класса, но и каждый определенный вне класса метод. Вот как выглядит функция `push()`:

```

template<class Type>
void Stack<Type>::push(Type var)
{
    st[++top] = var;
}

```

Имя `Stack<Type>` используется для того, чтобы идентифицировать класс, чьим методом является `push()`.

Для обычного, нешаблонного метода одного имени `Stack` было бы достаточно:

```

void Stack::push(int var) //Stack() как нешаблонная функция
{
    st[++top] = var;
}

```

но для шаблона функции приходится использовать еще и шаблонный аргумент: `Stack<Type>`.

Итак, имя шаблонного класса выглядит по-разному в зависимости от контекста. Внутри спецификации класса это просто само имя: `Stack`. Для методов, определенных вне классов, это имя класса и имя шаблонного аргумента: `Stack<Type>`. Когда вы определяете реальные объекты для хранения конкретного типа данных, именем шаблонного класса будет являться имя класса и имя типа: `Stack<float>`. Например:

```

class Stack                //Описатель класса Stack
{
};
void Stack<Type>::push(Type var) //определение push()
{
}
Stack<float> s1;           //объект типа Stack<float>

```

Нужно быть внимательным, чтобы корректно писать имена, соответствующие контексту. Очень просто можно забыть про `<Type>` или `<float>`, которые нужно добавлять к `Stack`. Компилятор не переносит таких ошибок.

Хотя это и не акцентировалось в предыдущем примере, но нужно всегда крайне аккуратно отслеживать правильность синтаксиса, когда метод возвращает значение своего собственного класса. Предположим, мы определили класс `Int`, который защищает законные права класса целых чисел (это обсуждалось в упражнении 4 главы 8 «Перегрузка операций»). Если бы вы использовали внешнее определение метода `xfunc()` данного класса, который возвращает значения типа `Int`, тогда пришлось бы писать `Int<Type>` и в качестве типа функции, и в качестве левой части оператора явного задания функции:

```
Int<Type> Int<Type>::xfunc(int arg)
{ }
```

Имя класса, являющегося типом аргумента функции, с другой стороны, вовсе не обязано включать в себя обозначение `<Type>`.

Создание класса связанных списков с помощью шаблонов

Рассмотрим еще один пример, в котором шаблоны помогают в создании классов-хранилищ данных. Для этого мы модифицируем программу `LINKLIST` из главы 10 «Указатели», которую советуем посмотреть сейчас еще разок. Требуется, чтобы не только класс `linklist` был переделан в шаблон, но и чтобы структура `link`, которая реально хранит каждый элемент данных, тоже стала шаблоном.

Листинг 14.5. Программа `TEMPLIST`

```
// templist.cpp
// Шаблон связанных списков
#include <iostream>
using namespace std;
////////////////////////////////////
template<class TYPE>          //структура link<TYPE>
struct link                   //элемент списка
// (внутри этой структуры определение link означает
// link<TYPE> )
{
    TYPE data;                //элемент данных
    link* next;               //указатель на следующий элемент
};
////////////////////////////////////
template<class TYPE>          //класс linklist<TYPE>
class linklist                //список ссылок
// (внутри этого класса linklist означает linklist<TYPE>)
{
private:
    link<TYPE>* first;        //указатель на первую ссылку
public:
    linklist()                //конструктор без аргументов
    { first = NULL; }         //первой ссылки нет
//примечание: можно вписать еще деструктор: для упрощения //листинга, не приведем его
//здесь.
    void additem(TYPE d);      //добавить данные (одна ссылка)
    void display();            //вывести все ссылки
};
```

Листинг 14.5 (продолжение)

```

////////////////////////////////////
template<class TYPE>
void linklist<TYPE>::additem(TYPE d)    //добавление данных
{
    link<TYPE>* newlink = new link<TYPE>; //создать новую
                                         // ссылку
    newlink->data = d;    //занести туда данные
    newlink->next = first; //то есть ссылку на следующий элемент
    first = newlink;     //теперь первая ссылка указывает на
                         //данную ссылку
}
//-----
template<class TYPE>
void linklist<TYPE>::display() //вывод всех ссылок
{
    link<TYPE>* current = first; //указатель - на первую ссылку
    while( current != NULL )     //выйти после последней ссылки
    {
        cout << endl << current->data; //вывести на экран
        current = current->next;       //сдвинуться на следующую ссылку
    }
}
//-----
int main()
{
    linklist<double> ld;    //ld - объект linklist<double>

    ld.additem(151.5);     //добавить три числа double в список ld
    ld.additem(262.6);
    ld.additem(373.7);
    ld.display();          //вывести весь список ld

    linklist<char> lch;    //lch - объект linklist<char>

    lch.additem('a');      //три символа (char) - в список lch
    lch.additem('b');
    lch.additem('c');
    lch.display();         //вывести весь список lch
    cout << endl;
    return 0;
}

```

В `main()` мы определяем два связанных списка: один — для хранения чисел типа `double`, другой — для хранения символов (типа `char`). В каждый из них заносим по три значения с помощью метода `additem()` и выводим все значения на экран с помощью `display()`. Результаты работы программы:

```

373.7
262.6
151.5
c
b
a

```

И класс `linklist`, и структура `link` пользуются шаблонным аргументом `TYPE` для подстановки любого типа данных (ну, конечно, не совсем любого: мы позднее

обсудим, какие типы реально могут использоваться). Таким образом, не только `linklist`, но и `link` должны быть именно шаблонами, начинающимися со строки `template<class TYPE>`

Вы обратили внимание на то, что не обязательно только классы или функции могут быть превращены в шаблоны? Такое волшебное превращение имеет право на существование для любых элементов программ, использующих изменяющиеся типы данных, как структура `link` в приведенном примере.

Как и прежде, необходимо следить за именованием класса (и структуры, в этом примере) в разных частях программы. Внутри его собственной спецификации мы используем только имя класса (или структуры): `linklist` (или `link`). Во внешних методах мы пишем имя класса, а за ним — имя шаблонного аргумента: `linklist<TYPE>`. Наконец, когда мы определяем объекты класса, мы уже используем конкретный нужный тип данных, что отражается на записи выражения:

```
linklist<double> ld;      //определяет объект ld класса
                          //linklist<double>
```

Хранение пользовательских типов

Нам до сих пор удавалось хранить с помощью шаблонных классов только базовые типы данных. В `TEMPLIST` мы хранили числа типа `double` и символы типа `char` в связанном списке. Возможно ли хранение пользовательских типов данных с использованием шаблонов? Ответ таков: можно, но осторожно.

Связный список работников

Рассмотрим класс `employee` из программы `EMPLOY` (глава 9 «Наследование»). О порожденных классах можно сейчас не беспокоиться. Мы займемся немного другим вопросом. Можно ли хранить объекты этого класса в связанном списке из примера `TEMPLIST`? Как и в случае шаблонных функций, надо определить, может ли шаблонный класс работать с объектами данного класса. Для этого нужно проверять, какие операции он выполняет над этими объектами. Класс `linklist` использует перегружаемую операцию вставки (`<<`) для вывода хранящихся в нем объектов:

```
void linklist<TYPE>::display()
{
    ...
    cout << endl << current->data; //используется <<
    ...
};
```

Никаких проблем с базовыми классами, для которых оператор вставки и так определен, не возникает. Но, к сожалению, в классе `employee` в программе `EMPLOY` этот оператор не перегружается. Поэтому придется поработать с классом и внести соответствующие изменения. Для упрощения процедуры получения данных о работниках от пользователя перегрузим-ка мы и оператор извлечения (`>>`). Данные, полученные в результате извлечения из потока, сохраним во временном объекте `emptemp`, прежде чем добавлять их в связанный список. Приведем листинг программы `TEMPLIST2`.

Листинг 14.6. Программы TEMLIST2

```

// temlist2.cpp
// связный список с использованием шаблона
// список, умеющий работать с классом employee
#include <iostream>
using namespace std;

const int LEN = 80;           //Максимальная длина имен
////////////////////////////////////
class employee                //класс employee
{
private:
    char name[LEN];           //имя работника
    unsigned long number;     //номер работника
public:
    friend istream& operator >> (istream& s, employee& e);
    friend ostream& operator << (ostream& s, employee& e);
};
//-----
istream& operator >> (istream& s, employee& e)
{
    cout << "\n Введите фамилию: "; cin >> e.name;
    cout << " Введите номер: ";   cin >> e.number;
    return s;
}
//-----
ostream& operator << (ostream& s, employee& e)
{
    cout << "\n Имя: " << e.name;
    cout << "\n Номер: " << e.number;
    return s;
}
////////////////////////////////////
template<class TYPE>          //структура "link<TYPE>"
struct link                   //один элемент списка
{
    TYPE data;                //элемент данных
    link* next;               //указатель на следующую ссылку
};
////////////////////////////////////
template<class TYPE>          //класс "linklist<TYPE>"
class linklist                //список ссылок
{
private:
    link<TYPE>* first;         //указатель на первую ссылку
public:
    linklist()                //конструктор без аргументов
        { first = NULL; }     //первой ссылки нет
    void additem(TYPE d);      //добавить данные (одну ссылку)
    void display();            //показать все ссылки
};
//-----
template<class TYPE>
void linklist<TYPE>::additem(TYPE d) //добавить данные
{
    link<TYPE>* newlink=new link<TYPE>; //создать новую ссылку
    newlink->data = d;              //внести в нее данные
}

```

```

newlink->next = first;           //указывающие на следующую ссылку
first = newlink; //теперь первая ссылка указывает на эту
}
//-----
template<class TYPE>
void linklist<TYPE>::display() //вывод всех ссылок
{
    link<TYPE>* current = first; //установить указатель на
                                //первую ссылку
    while( current != NULL )     //выйти после последней ссылки
    {
        cout << endl << current->data; //вывести данные
        current = current->next;       //перейти на следующую ссылку
    }
}
////////////////////////////////////
int main()
{
    linklist<employee> ltemp;      //ltemp - объект класса
    employee emptemp;              //"linklist<employee>"
    char ans;                      //временное хранилище объектов
    //ответ пользователя ('y' или 'n')

    do
    {
        cin >> emptemp;            //получить данные от пользователя
        ltemp.additem(emptemp);    //добавить в emptemp
        cout << "\nПродолжать (y/n)? ";
        cin >> ans;
    } while(ans != 'n');           //при окончании ввода
    ltemp.display();              //вывести весь связный список
    cout << endl;
    return 0;
}

```

В main() мы реализуем связный список ltemp. Затем в цикле просим пользователя ввести данные о работниках, после чего добавляем этого работника (то есть соответствующий объект) в список. Когда пользователь заканчивает ввод данных, выводим всю накопленную информацию. Вот один из примеров работы программы:

```

Введите фамилию: Рожанский
Введите номер:1233
Продолжать (y/n)? y
Введите фамилию: Штофф
Введите номер:2344
Продолжать (y/n)? y
Введите фамилию: Ершова
Введите номер:3455
Продолжать (y/n)? n

```

```

Имя: Ершова
Номер: 3455
Имя: Штофф
Номер: 2344
Имя: Рожанский
Номер: 1233

```

Обращаем ваше внимание на то, что класс `linklist` не нужно никак модифицировать, чтобы он хранил объекты пользовательского типа (`employee`). Вот в этом — вся прелесть шаблонных классов: они работают не только с базовыми типами, но и с типами, определенными пользователями!

Что можно хранить?

Мы уже выяснили, как определить, переменные какого типа могут храниться в шаблонном классе. Для этого нужно понять, какие операции выполняются в его методах. Вот, например, можно ли хранить в связанном списке (класс `linklist`) строки (класс `string`)? Методы класса используют операторы вставки (`<<`) и извлечения (`>>`), которые в отличных взаимоотношениях со строками, так что никаких препон для хранения строк не намечается. Но если в методах есть операции, несовместимые с каким-то типом, тогда значения этого типа хранить в данном классе нельзя.

UML и шаблоны

Шаблоны (или *параметризованные классы*) представляются на диаграммах классов UML в виде некоторых вариаций на тему символов классов. Имена аргументов шаблонов помещены в пунктирный прямоугольник, располагающийся рядом с верхним правым углом символа класса.

На рис. 14.3 показана диаграмма классов для программы `TEMPSTAK` (см. раздел «Шаблоны классов» данной главы).

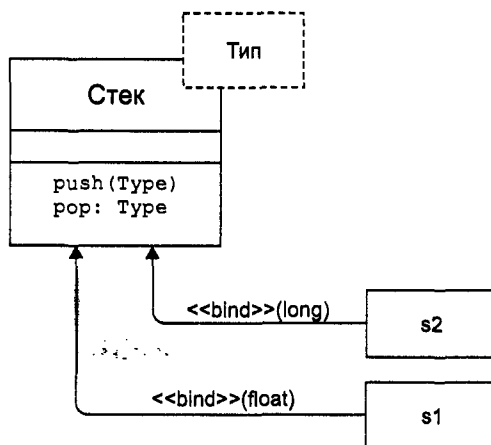


Рис. 14.3. Шаблон на диаграмме классов UML

У шаблона в этой программе только один аргумент — `Type`. Операции `push()` и `pop()` представлены своими типами возвращаемых значений (помните, что тип возвращаемого значения показывается после имени функции и отделяется от него двоеточием). Шаблонный аргумент обычно показывается прямо в обозначении операции, так `Type` показан на рисунке рядом с `push()` и `pop()`.

Диаграмма также показывает классы, реализуемые по шаблону: *s1*, *s2*.

Кроме шаблонов, на рис. 14.3 показаны две новые концепции UML: *зависимости* и *стереотипы*.

Зависимости в UML

Зависимость в UML называется отношение между двумя элементами, такое, что изменения в независимом элементе могут привести к изменениям в зависимом. Зависимый элемент, как следует из его названия, зависит от независимого, то есть использует его. Поэтому такие отношения иногда называются отношениями *использования*. В приведенном выше примере шаблонный класс является независимым элементом, а реализуемые классы — зависимыми.

Зависимость изображается на диаграмме пунктирной линией со стрелкой, направленной на независимый элемент. На рис. 14.3 реализованные по шаблону классы *s1* и *s2* являются зависимыми от шаблонного класса *Stack*, потому что изменения в последнем приводят или, по крайней мере, могут привести к изменениям в первых.

Вообще, зависимость — это довольно широкое понятие и применяется в UML во многих случаях. Собственно говоря, такие отношения, как ассоциация, обобщение и т. д. являются частными случаями отношения зависимости. Тем не менее они достаточно важны для того, чтобы их изображать на диаграммах в виде отдельных типов отношений.

Одним из характерных примеров ситуации, приводящей к отношению зависимости, является тот случай, когда один класс использует другой в качестве аргумента в своих операциях.

Стереотипы в UML

Стереотип — это способ сообщить дополнительные детали об элементе UML. Он представляется на диаграмме словом в двойных угловых скобках.

Например, пунктирная линия на рис. 14.3 представляет собой зависимость, но ничего не говорит о том, какого она рода-племени. Стереотип `<<bind>>` сообщает нам о том, что независимый элемент (шаблонный класс) реализует зависимый элемент (*s1*, например), используя указанные параметры, которые написаны в скобках после стереотипа. То есть в данном случае указывается, что `Type` будет заменен на `float`.

UML определяет множество разных стереотипов как элементы языка. Каждый из них применяется к своему элементу: классу, зависимостям и т. д. Можно добавить и что-нибудь свое.

Исключения

Исключения, второй большой вопрос данной главы, позволяют применить систематический, объектно-ориентированный подход к обработке возникающих в классах C++ ошибок. *Исключения* — это ошибки, возникающие во время работы программы. Они могут быть вызваны множеством различных обстоятельств,

таких, как выход за пределы памяти, ошибка открытия файла, попытка инициализировать объект недопустимым значением или использование индекса, выходящего за пределы вектора.

Для чего нужны исключения

Зачем нам понадобился еще один механизм обработки ошибок? Давайте посмотрим, как обрабатывались ошибки в прошлом. Программы на С имели привычку сообщать об ошибке возвращением установленного значения из функции, в которой она произошла. Например, функции работы с дисковыми файлами часто возвращают NULL или 0, чтобы показать, что произошла ошибка. Каждый раз при вызове этих функций происходит проверка возвращаемых значений:

```
if( somefunc() == ERROR_RETURN_VALUE )
    //обработка ошибки или вызов обработчика ошибок
else
    //нормальная работа
if( anotherfunc() == NULL )
    //обработка ошибки или вызов обработчика ошибок
else
    //нормальная работа
if( thirdfunc() == 0 )
    //обработка ошибки или вызов обработчика ошибок
else
    //нормальная работа
```

Одна из проблем данного подхода заключается в том, что каждый вызов такой функции должен проверяться программой. Окружение каждой функции if, добавление выражений обработки ошибок или вызова обработчиков приводит к тому, что код разрастается, как на дрожжах, становится запутанным, читается с трудом.

Проблема еще более усугубляется при использовании классов, так как ошибки могут возникать и при отсутствии явных вызовов функций. Например, представим себе, что какое-либо приложение определяет объекты класса:

```
someClass obj1, obj2, obj3;
```

Как объяснить программе, что произошла ошибка в конструкторе класса? Последний вызывается неявным образом, поэтому никакого возвращаемого значения от него не дождешься.

Еще больше краски сгущаются, когда в приложении используются библиотеки классов. Библиотека классов — это нечто, сделанное посторонними людьми, но используемое вами, программистами, купившими ее, в вашей программе. Поэтому очень сложно отследить ошибки и установить какие-то значения, которые сообщали бы об ошибках, — реализация библиотечных классов всегда остается неизвестной для программиста. Собственно говоря, проблема выявления ошибок, возникающих в глубинах библиотечных функций, — это, возможно, одна из самых глобальных задач, решаемых с помощью исключений. В конце этого раздела мы обязательно вернемся к этому вопросу.

Снова обратимся к исторической памяти программистов, пишущих на С. Они, должно быть, еще помнят те времена, когда в ходу был подход к обработке

ошибок с помощью пары функций `setjmp()` и `longjmp()`. Все бы хорошо, но для объектно-ориентированного программирования это не годится, поскольку таким способом невозможно должным образом обрабатывать деструкцию объектов.

Синтаксис исключений

Представим себе приложение, создающее объекты какого-либо класса и работающее с ними. В нормальной ситуации вызовы методов не приводят ни к каким ошибкам. Но иногда в программе возникает ошибка, которую обнаруживает метод. Он информирует приложение о случившемся. При использовании исключений это событие называется генерацией исключительной ситуации. В приложении при этом имеется отдельная секция кода, в которой содержатся операции по обработке ошибок. Этот код называют *обработчиком исключительных ситуаций* или *улавливающим блоком*. В нем отлавливаются исключения, сгенерированные методами. Любой код приложения, использующий объекты класса, заключается в *блок повторных попыток*. Соответственно, ошибки, возникшие в последнем, будут пойманы улавливающим блоком. Код, который не имеет никакого отношения к классу, не включается в блок повторных попыток. На рис. 14.4 показана схема, описанная выше.

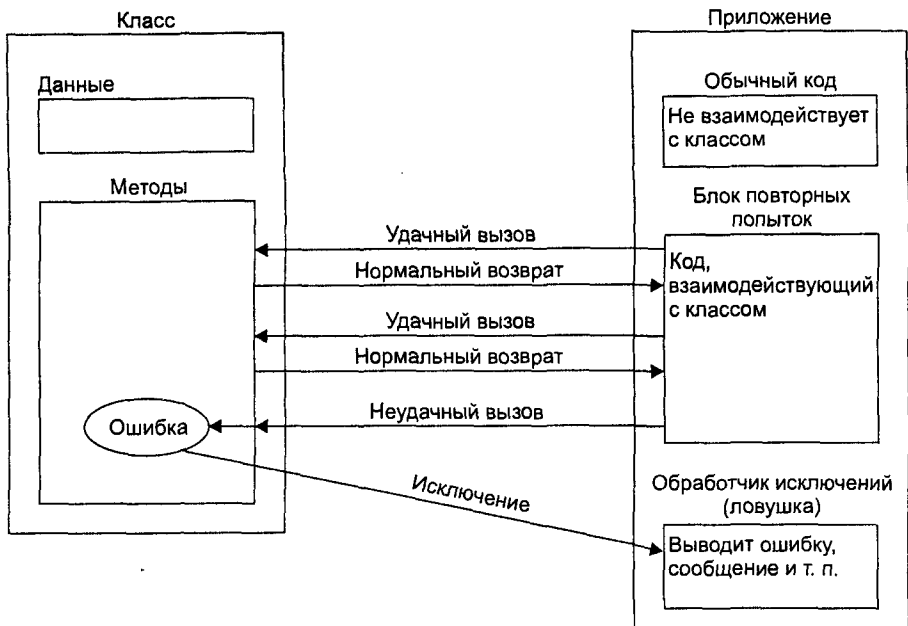


Рис. 14.4. Механизм исключений

Механизм исключений использует три новых слова C++: `catch`, `throw` и `try`. Кроме того, нужно создать новый тип сущности, называемый классом `exception`. Программа `XSYNTAX` — это еще не нормальная программа, а только скелет, приводимый для того, чтобы показать синтаксис.

Листинг 14.7. Скелет программы XSYNTAX

```
// xsyntax.cpp
// эта программа не работает!
////////////////////////////////////
class AClass          // просто класс
{
public:
    class AnError      // класс exception
    {
    };
    void Func()         //какой-то метод
    {
        if( /* условие ошибки */ )
            throw AnError(); // генерировать исключение
    }
};
////////////////////////////////////
int main()             // приложение как бы
{
    try                // блок повторных попыток
    {
        AClass obj1;   // взаимодействие с объектами AClass
        obj1.Func();   // тут может возникнуть ошибка
    }
    catch(AClass::AnError) //обработчик ошибок
    {                    // (улавливающий блок)
    }
    return 0;
}
```

Мы начинаем работу с создания класса AClass, представляющего собой некий класс, в котором могут возникнуть ошибки. Класс исключений, AnError, определен в общедоступной части класса AClass. Методы последнего подвергаются тщательному обследованию на предмет выявления ошибок. Если таковые обнаруживаются, мы с помощью ключевого слова throw и конструктора класса исключений производим генерацию исключительной ситуации:

```
throw AnError();          // 'throw' и конструктор класса AnError
```

В main() мы заключаем все выражения, каким-либо образом относящиеся к AClass, в блок повторных попыток. Если какие-то из них приводят к тому, что возникают подозрения на наличие ошибки в методе AClass, то генерируется исключение, и управление программой переходит к улавливающему блоку, который следует сразу же за блоком повторных попыток.

Простой пример исключения

Давайте, наконец, посмотрим на какую-нибудь реально работающую программу, использующую идею исключений. Пример, который вы сейчас увидите, является развитием программы STAKARAY из главы 7; в ней, если помните, создавалась стековая структура данных для хранения целочисленных значений. К сожалению, тот более ранний пример не обнаруживал двух важных ошибок. А именно: приложение могло пытаться извлечь или поместить в стек слишком много объ-

ектов, что привело бы либо к получению неправильных данных, либо к превышению размера массива. В программе XSTAK мы используем механизм исключений, чтобы обрабатывать эти ситуации.

Листинг 14.8. Программа XSTAK

```
// xstak.cpp
// Демонстрация механизма исключений
#include <iostream>
using namespace std;
const int MAX = 3;          //в стеке максимум 3 целых числа
////////////////////////////////////
class Stack
{
private:
    int st[MAX];            //стек: целочисленный массив
    int top;                //индекс вершины стека
public:
    class Range              //класс исключений для Stack
    {
        //внимание: тело класса пусто
    };
//-----
    Stack()                  //конструктор
    { top = -1; }
//-----
    void push(int var)
    {
        if(top >= MAX-1)     //если стек заполнен,
            throw Range();   //генерировать исключение
        st[++top] = var;     //внести число в стек
    }
//-----
    int pop()
    {
        if(top < 0)          //если стек пуст,
            throw Range();   //исключение
        return st[top--];    //взять число из стека
    }
};
////////////////////////////////////
int main()
{
    Stack s1;

    try
    {
        s1.push(11);
        s1.push(22);
        s1.push(33);
        // s1.push(44);      //Опаньки! Стек заполнен
        cout << "1: " << s1.pop() << endl;
        cout << "2: " << s1.pop() << endl;
        cout << "3: " << s1.pop() << endl;
        cout << "4: " << s1.pop() << endl;
        //Опаньки! Стек пуст
    }
}
```

Листинг 14.8 (продолжение)

```

catch(Stack::Range)    //обработчик
{
    cout << "Исключение: Стек переполнен или пуст"<<endl;
}
cout << "Приехали сюда после захвата исключения (или нормального выхода)" << endl;
return 0;
}

```

Имейте в виду, мы специально сделали стек таким маленьким, чтобы проще было привести пример возникновения исключительной ситуации, внося в него слишком много элементов.

Рассмотрим части этой программы, относящиеся к исключениям. Их четыре. В спецификации класса имеется класс исключений. Также там имеются выражения, генерирующие исключения. В `main()` есть часть кода, которая может привести к исключительным ситуациям, — это блок повторных попыток. Есть код, в котором эти ситуации обрабатываются, — улавливающий блок.

Описание класса исключений

В нашей программе класс исключений описывается **внутри класса Stack**:

```

class Range
{
    //Тело класса пусто!
};

```

Как видите, тело класса пусто, а потому его объекты не обладают ни данными, ни методами. Все, что нам на самом деле нужно в этой простенькой программе, это имя класса `Range`. Оно используется для связывания выражения генерации исключения с улавливающим блоком (классу не обязательно быть пустым, как мы увидим позднее).

Генерация исключения

В классе `Stack` исключение возникает, когда приложение пытается извлечь значение из пустого стека или занести значение в заполненный до отказа стек. Чтобы сообщить приложению о том, что оно выполнило недопустимую операцию с объектом класса `Stack`, методы этого класса проверяют указанные условия с использованием `if` и генерируют исключение, если эти условия выполняются. В программе `XSTAK` исключение генерируется в двух местах, оба раза с помощью выражения `throw Range();`

Второе слово этого незамысловатого предложения неявно запускает конструктор класса `Range` для создания, соответственно, объекта этого класса. Что касается первого слова, то оно передает управление программой обработчику ошибок (мы вскоре его увидим).

Блок повторных попыток (Try-блок)

Все выражения в `main()`, в которых могут произойти ошибки, то есть выражения, манипулирующие данными объектов класса `Stack`, заключены в фигурные скобки, перед которыми стоит ключевое слово `try`:

```
try
{
    //код, работающий с объектами, в которых могут
    //произойти ошибки
}
```

Это просто часть обычного кода программы. Такое могло бы понадобиться вне зависимости от того, используется механизм исключений или нет. В блок повторных попыток включаются только выражения, которым по долгу службы приходится иметь дело с классом **Stack**. Более того, в программе может быть несколько таких **try**-блоков, чтобы к этому классу был доступ из разных мест.

Обработчик ошибок (улавливающий блок)

Код, в котором содержатся операции по обработке ошибок, заключается в фигурные скобки и начинается с волшебного слова **catch**. Имя класса исключений должно включать в себя наименование класса, в котором он живет. В данном примере это **Stack::Range**.

```
catch(Stack::Range)
{
    //код-обработчик ошибок
}
```

Такая конструкция называется обработчиком прерываний. Она должна следовать непосредственно за блоком повторных попыток. Что касается приведенного примера, то в нем обработчик ошибок занимается только выводом на экран сообщения об ошибках, чтобы пользователь знал, почему его программа перестала работать. Управление программой «проваливается» на дно обработчика ошибок, так что программа может продолжать работу прямо с этого места. Обработчик, правда, может передать управление совсем в другое место, а еще чаще он поступает совсем просто: завершает работу программы.

Последовательность событий

Давайте подведем итоги и покажем последовательность действий программы при возникновении ошибки.

1. Код нормально выполняется вне блока повторных попыток.
2. Управление переходит в блок повторных попыток.
3. Какое-то выражение в этом блоке приводит к возникновению ошибки в методе.
4. Метод генерирует исключение.
5. Управление переходит к обработчику ошибок (улавливающему блоку), следующему сразу за блоком повторных попыток.

Вот и все. Обратите внимание, каким прозрачным и четким получился конечный алгоритм. В любом выражении блока повторных попыток может произойти ошибка, но у нас все под контролем, не нужно беспокоиться о получении возвращаемого значения для каждого из них, потому что это делается автоматиче-

ски. В данном конкретном примере мы сознательно написали два ошибочных выражения. Первое:

```
s1.push(44);           //в стек занесено слишком много элементов
```

приводит к исключительной ситуации (если в примере убрать комментарии перед ним), а второе:

```
cout << "4: " << s1.pop() << endl; //извлечение из пустого
                                     //стека
```

приводит к исключительной ситуации, если закомментировать первое выражение. Попробуйте сделать и так, и так. В обоих случаях будет выведено одно и то же сообщение:

Исключение: Стек переполнен или пуст

Многократные исключения

Можно спроектировать класс таким образом, чтобы он генерировал столько исключений, сколько нужно. Чтобы показать, как это делается, мы изменим программу XSTAK таким образом, чтобы она выдавала отдельные исключения при попытке переполнения стека и попытке извлечения из пустого стека.

Листинг 14.9. Программа XSTAK2

```
// xstak2.cpp
// Демонстрация обработчика двух исключений
#include <iostream>
using namespace std;
const int MAX = 3; //в стеке может быть до трех целых чисел
////////////////////////////////////////////////////////////
class Stack
{
private:
    int st[MAX];      //стек: массив целых чисел
    int top;          //индекс верхушки стека
public:
    class Full { };   //класс исключения
    class Empty { };  //класс исключения
//-----
    Stack()            //конструктор
    { top = -1; }
//-----
    void push(int var) //занести число в стек
    {
        if(top >= MAX-1) //если стек полон,
            throw Full(); //генерировать исключение Full
        st[++top] = var;
    }
//-----
    int pop()          //взять число из стека
    {
        if(top < 0)    //если стек пуст,
            throw Empty(); //генерировать исключение Empty
        return st[top--];
    }
};
```

```

    }
};
/////////////////////////////////////////////////////////////////
int main()
{
    Stack s1;

    try
    {
        s1.push(11);
        s1.push(22);
        s1.push(33);
        // s1.push(44);           //Опаньки: стек уже полон
        cout << "1: " << s1.pop() << endl;
        cout << "2: " << s1.pop() << endl;
        cout << "3: " << s1.pop() << endl;
        cout << "4: " << s1.pop() << endl;
        //Опаньки: стек пуст
    }
    catch(Stack::Full)
    {
        cout << "Ошибка: переполнение стека" << endl;
    }
    catch(Stack::Empty)
    {
        cout << "Ошибка: стек пуст" << endl;
    }
    return 0;
}

```

В этой программе были описаны два класса исключений:

```

class Full { };
class Empty { };

```

Выражение

```
throw Full();
```

исполняется, если программа вызывает `push()` при уже заполненном стеке, а
`throw Empty();`

исполняется, если программа вызывает `pop()` при пустом стеке.

Для каждого исключения используется отдельный отлавливающий блок:

```

try
{
    //код, работающий с объектами класса Stack
}
catch
{
    //код обработки исключения Full
}
catch
{
    //код обработки исключения Empty
}

```

Все отлавливающие блоки, используемые с конкретными блоками повторных попыток, должны следовать в коде непосредственно за ними. В этом случае каждое исключение приведет к выводу своего сообщения об ошибке: «Стек заполнен» или «Стек пуст». Дашь каждому отлавливающему блоку по своему исключению! Таков девиз приведенной выше программы, да так, собственно, зачастую и делается в программах. Группа отлавливающих блоков (*цепь ловушек*) работает примерно как выражение `switch` в том смысле, что исполняется только один соответствующий блок кода. После обработки исключения управление программой переходит к тому месту программы, которое следует за всеми блоками-ловушками. (В отличие от `switch`, не нужно заканчивать каждый отлавливающий блок `break`. В какой-то мере эти блоки напоминают функции.)

Исключения и класс `Distance`

В качестве еще одного примера технологии исключений рассмотрим ее применение к уже родному классу `Distance`. Объекты этого класса по-прежнему состоят из целочисленных значений футов и значений дюймов типа `float`. При этом число дюймов не должно превышать 12,0. Проблема этого класса в предыдущих программах заключалась в том, что невозможно было защититься от ввода пользователем неправильного значения дюймов. Это могло приводить к некорректности данных, получаемых в результате выполнения арифметических операций, так как в этих операциях предполагалось, что значения дюймов все-таки меньше 12,0. Недопустимые значения тоже могли быть выведены, а пользователь мог впасть в легкий ступор наедине с форматом `7'-15"`.

Перепишем теперь класс `Distance` таким образом, чтобы можно было использовать механизм исключений для обработки ошибок.

Листинг 14.10. Программа `XDIST`

```
// xdist.cpp
// Исключения и класс Distance
#include <iostream>
using namespace std;
////////////////////////////////////
class Distance          //класс английских мер расстояний
{
private:
    int feet;
    float inches;
public:
    class InchesEx { }; //класс исключений
//-----
    Distance()           //конструктор (без аргументов)
    { feet = 0; inches = 0.0; }
//-----
    Distance(int ft, float in) //конструктор (2 arg-та)
    {
        if(in >= 12.0)           //если дюймы указаны неверно.
            throw InchesEx();    //генерировать исключение
        feet = ft;
        inches = in;
    }
}
```

```
//-----
void getdist()           //получить длину от пользователя
{
    cout << "\nEnter feet: "; cin >> feet;
    cout << "Enter inches: "; cin >> inches;
    if(inches >= 12.0) //если дюймы неправильные.
        throw InchesEx(); //генерировать исключение
}
//-----
void showdist()          //вывод расстояний
{ cout << feet << "\'-" << inches << '\\"'; }
};
////////////////////////////////////
int main()
{
    try
    {
        Distance dist1(17, 3.5); //конструктор (2 аргумента)
        Distance dist2;          //конструктор (без аргументов)
        dist2.getdist();          //получить расстояние
                                   //вывести расстояния
        cout << "\ndist1 = "; dist1.showdist();
        cout << "\ndist2 = "; dist2.showdist();
    }
    catch(Distance::InchesEx) //поймать исключения
    {
        cout << "\nОшибка инициализации: "
              << "значение дюймов превышает предельно допустимое.";
    }
    cout << endl;
    return 0;
}
```

Итак, разберемся с этой программой. Здесь мы создали класс исключений, называемый `InchesEx`, для класса `Distance`. При любой попытке пользователя ввести неправильное значение генерируется исключение. Так случается в двух местах: в конструкторе аргументов, где программист может неверно задать инициализирующие значения, а также в функции `getdist()`, где пользователь может ввести ошибочные данные в ответ на просьбу ввести дюймы. Можно проверять таким способом, например, случайный ввод отрицательных значений, да и другие ошибки ввода.

В `main()` все взаимодействия с объектами `Distance` заключены в блок повторных попыток, а блок-ловушка выводит сообщение об ошибках.

В серьезно разрабатываемых программах, конечно, можно обрабатывать ошибки пользователя (как бы противопоставляя их ошибкам программиста) по-другому. Более приемлемо, с точки зрения пользователя, было бы, например, не выходить из программы и не просто выводить строку сообщения об ошибке, а возвращаться на начало блока повторных попыток. Собственного говоря, это блок так и называется из-за того, что запуск программы после сбоя в общем случае может производиться с него. Так вот, если вернуться к началу этого блока повторных попыток, то пользователю тем самым может быть дан шанс ввести правильное значение дюймов.

Исключения с аргументами

А что, если приложению нужно больше информации о том, что привело к возникновению исключительно ситуации? Например, в программе XDIST может понадобиться выяснить, какие именно значения дюймов были введены. Далее, если, например, одно и то же исключение генерируется разными методами класса, то бывает важно выяснить, кто же виновник того, что произошла ошибка. Может быть, существует какой-то способ передать такую информацию от метода, где было сгенерировано исключение, к приложению, которое его поймало?

На этот вопрос можно ответить, если вспомнить, что генерация исключения включает в себя не только передачу управления обработчику, но и создание объекта класса исключений с помощью вызова его конструктора. Например, в XDIST мы создаем объект типа InchesEx при генерации исключения с помощью выражения

```
throw InchesEx();
```

Если мы добавляем компонентные данные к классу исключений, мы можем их инициализировать при создании объекта. Обработчик исключений затем может извлечь данные из этого объекта при поимке исключения. С таким же успехом можно сделать какую-нибудь надпись на футбольном мяче и перекинуть его приятелю через забор. Немного модифицируем программу XDIST, чтобы продемонстрировать вышесказанное.

Листинг 14.11. Программа XDIST2

```
// xdist2.cpp
// исключения с аргументами
#include <iostream>
#include <string>
using namespace std;
////////////////////////////////////
class Distance          //класс английских расстояний
{
private:
    int feet;
    float inches;
public:
//-----
    class InchesEx      //класс исключений
    {
    public:
        string origin;  //для имени функции
        float iValue;   //для хранения ошибочного
                        //значения

        InchesEx(string or, float in) //конструктор с
                                    //двумя аргументами
        {
            origin = or;             //сохраненная строка
                                    //с именем виновника ошибки
            iValue = in;             //сохраненное неправильно
                                    //значение дюймов
        }
    };
//конец класса исключений
```

```
//-----
Distance()           //конструктор (без аргументов)
{ feet = 0; inches = 0.0; }
//-----
Distance(int ft, float in) //конструктор (2 аргумента)
{
    if(in >= 12.0)
        throw InchesEx("Конструктор с двумя аргументами", in);
    feet = ft;
    inches = in;
}
//-----
void getdist()        //получить данные от пользователя
{
    cout << "\nВведите футы: "; cin >> feet;
    cout << "Введите дюймы: "; cin >> inches;
    if(inches >= 12.0)
        throw InchesEx("функция getdist()", inches);
}
//-----
void showdist()        //вывести расстояние
{ cout << feet << "\'-' << inches << '\'"; }
};
////////////////////////////////////
int main()
{
    try
    {
        Distance dist1(17, 3.5); //конструктор с двумя
                                //аргументами
        Distance dist2;          //конструктор без аргументов
        dist2.getdist();          //получить значение
                                //вывести расстояния
        cout << "\ndist1 = "; dist1.showdist();
        cout << "\ndist2 = "; dist2.showdist();
    }
    catch(Distance::InchesEx ix) //обработчик ошибок
    {
        cout << "\nОшибка инициализации. Виновник: " << ix.origin
            << "\n Введенное значение дюймов " << ix.iValue
            << " слишком большое.";
    }
    cout << endl;
    return 0;
}
```

При передаче данных в обработчик ошибок производятся три действия: задание методов и конструктора класса исключений, инициализация этого конструктора при генерации исключения, обращение к данным объекта при поимке исключения. Давайте разберемся во всем по порядку.

Спецификация данных в классе исключений

Данные в классе исключений лучше всего сделать общедоступными, чтобы обработчик имел прямой доступ к ним. Вот так выглядит спецификация класса исключений InchesEx в программе XDIST2:

```

class InchesEx                                //класс исключений
{
public:
    string origin;                            //для имени функции
    float iValue;                             //для хранения ошибочного
                                              //значения
InchesEx(string or, float in)                //конструктор с
                                              //двумя аргументами
{
    origin = or;                             //сохраненная строка
                                              //с именем виновника ошибки
    iValue = in;                             //сохраненное неправильное
                                              //значение дюймов
}
};                                             //конец класса исключений

```

В этом классе имеются общедоступные переменные для объекта класса `string`, в котором будут содержаться сведения об имени вызываемого метода, а также переменные типа `float` для хранения ошибочных значений дюймов.

Инициализация объекта класса исключений

Как же инициализировать данные при генерации исключительной ситуации? В конструкторе с двумя аргументами класса `Stack` мы пишем:

```
throw InchesEx("Конструктор с двумя аргументами". in);
```

А в методе `getdist()` для класса `Stack` пишем:

```
throw InchesEx("функция getdist()". inches);
```

При генерации исключения обработчик выведет соответствующую строку и значение количества дюймов. Из строки мы узнаем, в каком методе произошла ошибка, а `inches` сообщит, какое именно значение ошибочно ввел пользователь. Такой способ обработки исключений позволит программисту или пользователю более точно определить причину сбоя в программе.

Извлечение данных из объекта класса исключений

Как нам добыть данные при поимке исключения? Проще всего сделать их общедоступной частью класса исключений, как мы и сделали в нашем примере. В блоке-ловушке можно объявить `ix` как имя объекта класса исключений, за которым мы охотимся с нашим большим исключительным ружьем. Используя это имя, можно запросто обращаться к данным объекта, записывая их через точку:

```

catch(Distance::InchesEx ix)
{
    //прямой доступ к 'ix.origin' и 'ix.value'
}

```

Можно совершенно спокойно выводить на экран значения `ix.origin` и `ix.value`. Приведем пример взаимодействия пользователя с программой `XDIST2`. Пользователь ввел, как обычно, некорректное значение дюймов, оно превышает значение 12,0.

```

Введите футы: 7
Введите дюймы: 13.5

```

Ошибка инициализации. Виновник: функция `getdixt()`
Введенное значение дюймов 13.5 слишком большое.

Если же нерадивому программисту вдруг захочется поменять определение `dist1` в `main()` на

```
Distance dist1(17. 22.25);
```

сгенерируется исключение, в результате чего на экране появится надпись следующего содержания:

Ошибка инициализации. Виновник: конструктор с двумя аргументами.
Введенное значение дюймов 22.25 слишком большое.

Конечно, с аргументами исключений можно делать все, что угодно, но типичное их использование — это получение дополнительной информации, помогающей диагностировать ошибки.

Класс `bad_alloc`

В стандарт C++ включено несколько встроенных классов исключений. Наверное, чаще всего в работе используется класс `bad_alloc`, который генерирует исключение, если возникает ошибка резервирования памяти при использовании `new`. В более ранних версиях C++ это исключение называлось `xalloc`. На момент публикации этой книги прежний подход все еще использовался в Microsoft Visual C++. Если правильно установить блок-ловушку и блок повторных попыток, то `bad_alloc` можно использовать совершенно без проблем. В следующем коротеньком примере `BADALLOC` показано, как это делается.

Листинг 14.12. Программа `BADALLOC`

```
// badalloc.cpp
// Демонстрация исключения bad_alloc
#include <iostream>
using namespace std;

int main()
{
    const unsigned long SIZE = 10000; //объем памяти
    char* ptr;                        //указатель на адрес в памяти

    try
    {
        ptr = new char[SIZE]; //разместить в памяти SIZE байт
    }
    catch(bad_alloc)           //обработчик исключений
    {
        cout << "\n Исключение bad_alloc: невозможно разместить данные в памяти.\n";
        return(1);
    }
    delete[] ptr;              //освободить память
    cout << "\nПамять используется без сбоев.\n";
    return 0;
}
```

Все выражения, каким-либо образом связанные с `new`, поместите в блок повторных попыток. Блок-ловушка, который ставится сразу за этим блоком, обрабатывает ошибку, то есть выведет сообщение и закроет программу.

Размышления об исключениях

Мы показали только простейшие примеры использования механизма исключений. Углубляться в данную тему нам не позволяет объем книги, но хочется высказать напоследок несколько соображений относительно этой технологии.

Вложенные функции

Выражение, приводящее к исключительной ситуации, не обязательно располагается непосредственно в блоке повторных попыток. Оно может быть и в функции, вызываемой выражением из этого блока (или функцией, вызванной функцией, вызванной функцией из блока повторных попыток, и т. д., понятное дело). Поэтому следует сделать блок повторных попыток верхним уровнем программы. О функциях, находящихся на более низких уровнях, можно заботиться не столь усердно, поскольку в любом случае они вызываются с верхнего уровня, то есть из блока повторных попыток (тем не менее функциям среднего уровня иногда бывает полезно добавлять данные о себе в исключения или передавать их вниз по иерархии).

Исключения и библиотеки классов

Важной проблемой, которую решает механизм исключений, является выявление ошибок в библиотеках классов. В библиотечной функции легко может возникнуть ошибка, но обычно совершенно неясно, где именно и что с ней делать. Кроме того, надо учитывать, что алгоритмы для библиотеки были написаны какими-то другими людьми, в другое время, то есть довольно сильно могут отличаться по концепции от программы, которую вы собираетесь писать. Что обязана в любом случае библиотечная функция сделать, так это сообщить о возникшей ошибке в вызвавшую ее программу. Проще говоря, библиотечная функция должна сказать: «У меня произошла ошибка. Я не знаю, что с ней делать, но она уже произошла» и вызывающая программа будет обрабатывать эту ошибку по своему усмотрению.

Механизм исключений включает в себя такую возможность, поскольку исключения передаются вверх по иерархической лестнице вложенных функций до тех пор, пока не дойдут до блока-ловушки. Выражение выдачи исключения может входить в библиотеку, но ловушка может быть только в вызывающей программе, которая одна знает, как лучше обработать ошибку.

Если вы взяли за написание библиотеки классов, позаботьтесь о том, чтобы исключения выдавались в вызывающую программу из всех функций, в которых может случиться что-то неординарное. Если же вы пишете программу, использующую библиотеку, то нужно предусмотреть блок повторных попыток и блок-ловушку для любых исключений, которые может выдавать библиотека.

Исключения — не панацея

Механизм исключений невозможно применять ко всем типам ошибок. Есть некий потолок, касающийся размеров программы и времени на обработку ошибок. Например, все-таки, наверное, нецелесообразно использовать исключения для работы с ошибками ввода (например, ввода символов вместо цифр), такие коллизии гораздо проще ликвидировать традиционными способами. Да-да, просто в цикле проверять, что вводит пользователь, и принимать соответствующие решения. В случае необходимости — просить пользователя ввести данные заново.

Автоматически вызываемые деструкторы

Технология обработки исключений удивительно умна и практична. При генерации исключения деструктор вызывается автоматически для того объекта, который был создан в коде как раз перед тем местом, где возникла ошибка. Это действительно необходимо, поскольку приложение не знает, какое выражение стало причиной исключения, а если программа хочет после обработки ошибки продолжать свое выполнение, то ей придется (по меньшей мере) пройти заново весь блок повторных попыток. Механизм исключений гарантирует, что код в try-блоке будет «перезагружен».

Обработка исключений

После успешной ловли исключений вам, как истинному охотнику, вероятно, захочется прикончить всю программу, чтобы никаких исключений больше тут не водилось вообще. Механизм исключений позволяет выявить виновника произошедшей ошибки, представить его имя пользователю, а перед завершением программой произвести все необходимые ритуалы очистки памяти, завершения стартовавших процессов и т. п. Лучшее средство для очистки совести компьютера перед закрытием приложения — это деструктор. Как раз с его помощью можно вывести даже самые невыводимые пятна в памяти компьютера, во всяком случае, той ее части, которая использовалась объектами.

В других ситуациях у вас может не появиться намерения завершить программу. Может быть, ошибку можно исправить «на лету» или попросить пользователя ввести данные корректно и исчерпать тем самым инцидент. В таком случае, блоки повторных попыток и блоки-ловушки замыкаются в цикл так, чтобы после работы обработчика исключений управление в программе вернулось бы к началу блока повторных попыток.

Если не нашлось обработчика для сгенерировавшегося исключения, операционная система бесцеремонно закрывает программу.

Резюме

Шаблоны позволяют создавать семейства функций или семейства классов, выполняющих одни и те же операции с разными типами данных. Если вы вдруг ловите себя на том, что пишете функции, которые делают одно и то же, но с разными типами данных, значит, пришла пора написать шаблон функции. Если вы

ловите себя на том, что пишете уже в который раз спецификацию класса, который делает то же, что и предыдущие, но с другими типами, следует подумать о написании шаблона класса. Этим вы сохраните свое драгоценное время, а в результате получится ясный, понятный и компактный код, причем алгоритмы можно будет менять очень легко и сразу для работы со всеми используемыми типами данных.

Исключения — это механизм обработки ошибок, возникающих при некорректной работе программ, написанных на C++, с использованием систематического, объектно-ориентированного подхода. Исключительные ситуации обычно возникают вследствие ошибочных выражений в блоках повторных попыток (try-блоках), которые работают с объектами некоторого класса. Метод класса обнаруживает ошибку и «генерирует исключение», которое ловится программой с помощью класса, следующего в коде обработчика исключений сразу за блоком повторных попыток.

Вопросы

Ответы на эти вопросы можно найти в приложении Ж.

1. Шаблоны позволяют удобным способом создавать семейства:
 - а) переменных;
 - б) функций;
 - в) классов;
 - г) программ.
2. Шаблонный аргумент всегда начинается с ключевого слова _____.
3. Истинно ли утверждение о том, что шаблоны автоматически создают разные версии класса в зависимости от введенных пользователем данных?
4. Напишите шаблон функции, всегда возвращающей свой аргумент, умноженный на два.
5. Шаблонный класс:
 - а) создается для того, чтобы храниться в разных контейнерах;
 - б) работает с разными типами данных;
 - в) генерирует идентичные объекты;
 - г) генерирует классы с различным числом методов.
6. Истинно ли утверждение о том, что шаблон может иметь несколько аргументов?
7. Создание реальной функции из шаблона называется _____ функцией.
8. Реальный код шаблонной функции генерируется при:
 - а) объявлении функции в исходном коде;
 - б) определении функции в исходном коде;

- в) вызове функции в исходном коде;
 - г) запуске функции во время работы программы.
9. Ключевой для шаблонов является концепция, согласно которой _____ заменяется на имя, которое подставляется вместо _____.
10. Шаблоны часто используются с классами, которые _____.
11. Исключение в большинстве случаев возникает из-за:
- а) программиста, написавшего исходный код приложения;
 - б) создателя класса, написавшего его методы;
 - в) ошибки выполнения;
 - г) сбоя в операционной системе.
12. При работе с механизмом исключений в C++ используются следующие ключевые слова: _____, _____, _____.
13. Напишите выражение, генерирующее исключение, используя класс `Bounds-Error` (тело класса пусто).
14. Истинно ли утверждение о том, что выражения, которые могут создать исключительную ситуацию, должны быть частью блока-ловушки?
15. Исключения передаются:
- а) из блока-ловушки в блок повторных попыток;
 - б) из выражения, создавшего исключительную ситуацию, в блок повторных попыток;
 - в) из точки, где возникла ошибка, в блок-ловушку;
 - г) из выражения, в котором возникла ошибка, в блок-ловушку.
16. Напишите спецификацию класса исключений, хранящего номер ошибки и ее название. Включите в класс конструктор.
17. Истинно ли утверждение о том, что выражение, генерирующее исключение, не должно быть расположено в блоке повторных попыток?
18. Для следующих ошибок обычно генерируется исключение:
- а) чрезмерное количество данных грозит переполнить массив;
 - б) пользователь нажал `Ctrl+C` для закрытия программы;
 - в) скачок напряжения в сети привел к перезагрузке системы;
 - г) `new` не может зарезервировать необходимый объем памяти.
19. Дополнительная информация, передающаяся при генерации исключения, может быть помещена в:
- а) ключевое слово `throw`;
 - б) функцию, вызвавшую ошибку;
 - в) блок-ловушку;
 - г) объект класса исключений.
20. Истинно ли утверждение о том, что программа может продолжить свое выполнение после возникновения исключительной ситуации?

21. Говоря о зависимостях, мы под шаблонным классом подразумеваем _____ элемент программы, а под реализуемым — _____.
22. Шаблонный класс показывается на диаграмме UML в виде:
- а) обычного класса с каким-то аппендиксом;
 - б) пунктирной линии;
 - в) прямоугольника с пунктирным контуром;
 - г) нет правильного ответа.
23. Истинно ли утверждение о том, что зависимость — это вид ассоциации?
24. Стереотип дает _____ об элементе UML.

Упражнения

Решения к упражнениям, помеченным знаком *, можно найти в приложении Ж.

- *1. Напишите шаблон функции, возвращающей среднее арифметическое всех элементов массива. Аргументами функции должны быть имя и размер массива (типа `int`). В `main()` проверьте работу функции с массивами типа `int`, `long`, `double` и `char`.

- *2. Очередь — это тип хранилища данных. Она напоминает по своей сути стек, только вместо правила LIFO (последний вошел — первый вышел) использует правило FIFO (первый вошел — первый вышел). В общем, как очередь у окошка в банке. Если вы заносите в массив числа 1, 2, 3, обратно вы их получите в той же последовательности.

Стеку требуется для работы только один индекс массива (`top`, вершина массива. См. программу `STAKARAY` из главы 7). Очереди же требуется два индекса: индекс хвоста, куда добавляются новые элементы, и индекс головы очереди, откуда исчезают старые. Хвост постепенно сдвигается, как и голова. Если хвост или голова достигает конца массива, он переставляется на начало.

Напишите шаблон класса для работы с очередью. Предположим, что программист не будет совершать ошибок при написании ее модели. Например, вместимость очереди не будет превышена, а из пустой очереди не будет производиться попыток удаления данных. Определите несколько очередей разных типов и поработайте с их данными.

- *3. Добавьте механизм обработки исключений в программу из упражнения 2. Рассмотрите два исключения: при превышении размера очереди и при попытке удаления данных из пустой очереди. Это можно сделать, добавив новый элемент данных в класс очереди — счетчик текущего числа элементов в очереди. Инкрементируйте счетчик при добавлении нового элемента, декрементируйте, соответственно, при удалении элемента из головы очереди. Генерируйте исключение, если счетчик превысил размер массива или если он стал меньше 0.

Можно попробовать сделать `main()` интерактивной, чтобы пользователь мог вводить и извлекать данные. Так будет проще проверить работу написанных функций. С помощью механизма исключений программа должна обеспечивать повторную попытку ввода данных пользователем без нарушения целостности содержимого очереди.

4. Создайте функцию `swaps()`, обменивающую значения двух аргументов, посылаемых ей. (Возможно, вы догадаетесь передавать эти аргументы по ссылке.) Сделайте из функции шаблон, чтобы она могла использоваться с любыми числовыми типами данных (`int`, `char`, `float` и т. д.). Напишите `main()` для тестирования функции.
5. Создайте функцию `amax()`, возвращающую значение максимального элемента массива. Аргументами функции должны быть адрес и размер массива. Сделайте из функции шаблон, чтобы она могла работать с массивом любого числового типа. Напишите секцию `main()`, в которой проверьте работу функции с разными типами массивов.
6. Начните работу с класса `safearray` из программы `ARROVER3` (глава 8). Сделайте класс шаблоном, чтобы массив мог хранить любые типы данных. В секции `main()` создайте, по крайней мере, два массива разных типов.
7. За основу возьмите класс `frac` и калькулятор с четырьмя функциями из упражнения 7 главы 8. Сделайте этот класс шаблоном, чтобы его можно было реализовывать с использованием различных типов данных в качестве делимого и делителя. Конечно, это должны быть целочисленные типы, что строго ограничивает вас в их выборе (`char`, `short`, `int` и `long`). Можно, впрочем, определить и свой целочисленный класс, никто не запрещает. В `main()` реализуйте класс `frac<char>` и используйте его при разработке калькулятора с четырьмя функциями. Этому классу требуется меньше памяти, чем `frac<int>`, но с его помощью невозможно выполнять деление больших чисел.
8. Добавьте класс исключений к программе `ARROVER` из главы 8, чтобы индексы, выходящие за пределы массива, вызывали генерацию исключения. Блок-ловушка может выводить пользователю сообщение об ошибке.
9. Измените программу из предыдущего упражнения таким образом, чтобы в сообщении об ошибке входила информация о значении индекса, приведшего к сбою.
10. Есть разные мнения на тему того, когда целесообразно использовать механизм исключений. Обратимся к программе `ENGLERR` из главы 12 «Потоки и файлы». Надо ли в виде исключений оформлять ошибки, которые делает пользователь при вводе данных? Для этого упражнения будем считать, что надо. Добавьте класс исключений к классу `Distance` в указанной программе (см. также примеры `XDIST` и `XDIST2` в этой главе). Сгенерируйте исключения во всех местах, где `ENGLERR` выводила сообщение об ошибке. Для выявления конкретной причины исключения (вместо дюймов введены какие-то символы, значение дюймов выходит за область допустимых

значений и т. д.). Кроме того, исключения должны генерироваться в качестве реакции на ошибки, возникающие в функции `isint()` (ничего не было введено, введено слишком много разрядов значений, введен символ вместо числа, число выходит за область допустимых значений). Вопрос к вам: если функция `isint()` сгенерировала исключение, может ли она оставаться независимой?

Можно зациклить блок повторных попыток и блок-ловушку (создать цикл `do`), чтобы после обработки исключения программа продолжала работу, предлагая пользователю повторно ввести данные.

Если еще немного подумать, то может возникнуть желание генерировать исключение в конструкторе с двумя аргументами (в том случае, если программист инициализирует `Distance` значениями, выходящими за область допустимых).

11. За основу возьмите программу `STRPLUS` из главы 8. Добавьте класс исключений, генерируйте исключения в конструкторе с одним аргументом в случае, если строка инициализации слишком длинная. Генерируйте еще одно исключение в перегруженном операторе «+», если результат конкатенации оказывается слишком длинным. Сообщайте пользователю о том, какая именно ошибка произошла.
12. Иногда проще всего бывает использовать механизм исключений, если создать новый класс, чьим компонентом является класс исключений. Попробуйте проделать такую операцию с классом, использующим исключения для обработки файловых ошибок. Создайте класс `dofile`, включающий в себя класс исключений и методы чтения, и записи файлов. Конструктор этого класса в качестве аргумента может брать имя файла, а действием конструктора, соответственно, может быть открытие этого файла. Можно сделать так, чтобы метод переставлял указатель позиции в начало файла. Используйте программу `REWERR` из главы 12 в качестве примера, напишите секцию `main()`, в которой сохранилась бы функциональность прежней версии, но использовались бы методы класса `dofile`.

Р. ЛАФОРЕ

ОБЪЕКТНО-ОРИЕНТИРОВАННОЕ ПРОГРАММИРОВАНИЕ В C++

4-Е ИЗДАНИЕ

КНИГИ, КОТОРЫЕ НЕ СТАРЕЮТ!

КЛАССИКА COMPUTER SCIENCE

Изучение объектно-ориентированной парадигмы программирования обычно ассоциируется с многочасовым курсом лекций и утомительными лабораторными занятиями, во время которых вас бросает из крайности в крайность — от тупого отчаяния до ложного ощущения полного понимания. Книга популярного среди американских студентов писателя Роберта Лафоре призвана помочь изучающему C++ избежать таких нагрузок на психику и без чьей-либо помощи пройти путь от застенчивого ученика до уверенного в своих силах программиста.

Столь значительный объем этого учебника обусловлен как сложностью предмета, так и стремлением автора не оставить белых пятен в программистском образовании читателя: любой элемент теории немедленно иллюстрируется небольшим примером кода, для приобретения практических навыков даются детально прокомментированные тексты полномасштабных приложений, а для закрепления полученных знаний читателю предлагается выполнить множество самостоятельных упражнений.

ISBN 5-94723-302-9



9 785947 123302

Обсудите наш веб-сайт: www.piter.com

 ПИТЕР
WWW.PITER.COM

 SAMS