Лекция 10. Шаблоны и исключения

Шаблоны являются инструментом ООП, позволяющим использовать одни и те же функции или классы для обработки разных типов данных. Концепция шаблонов может быть реализована как по отношению к функциям, так и по отношению к классам.

Допустим, требуется написать функцию вычисления модуля чисел.

```
int abs(int n)

{

if (n<0) return –n;

elsereturnn;

}
```

Описанная функция берет аргумент типа int и возвращает результат того же типа. Если нужно найти модуль числа типа float, то придется писать ещё одну функцию:

```
float abs(float n)

{

if (n<0) return –n;

elsereturnn;

}
```

Тело функций при этом ничем не отличается. Эти функции могут быть перегружены и иметь одинаковые имена, но все равно для каждой из них нужно писать отдельное определение. Многократное переписывание таких функций-близнецов утомляет и способствует порождению ошибок. А если где-то нужно исправить алгоритм, придется исправлять его в теле каждой функции.

Шаблоны функций в C++ как раз и существуют для того, чтобы можно было написать алгоритм всего один раз и заставить его работать с различными типами данных возвращая результаты разного типа.

Следующий пример показывает, как пишется шаблон функции, вычисляющей модуль числа и как он потом используется.

```
template<class T> // Это шаблон функции

T abs(T n ) //

{ //

if (n<0) return –n; //

else return n; //
```

} //

int main

{

int a = –10, b;

float x = 3, y;

b = abs(a); // теперь функция abs( ) может работать с любым типом данных

y = abs(x);

}

Заданная таким образом функция abs( ) может работать с любыми типами данных, если для них определен оператор < и унарный оператор -. Типы данных определяются функцией при передаче аргумента.

Сутью концепции шаблонов функций является представление использующегося функцией типа не в виде какого-то специфического, а с помощью названия, вместо которого может быть подставлен любой тип. Ключевое слово template сообщает компилятору о том, что определяется шаблон функции.

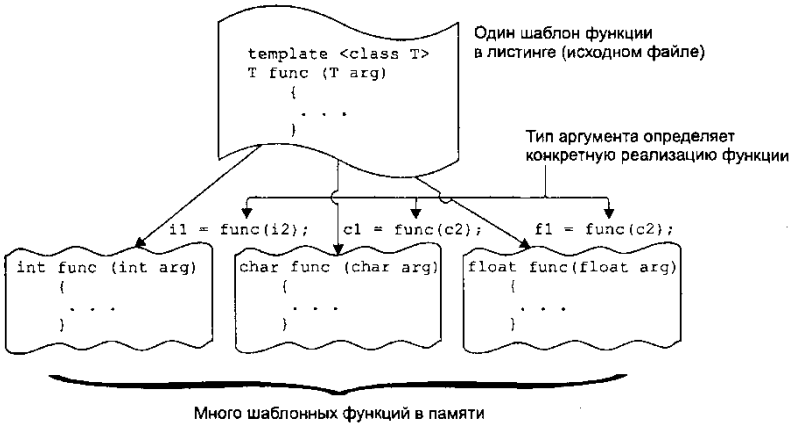

Рисунок 1. Сущность шаблона функции.

Генерация кода при определении шаблона не происходит до тех пор, пока функция не будет реально вызвана в ходе исполнения программы. Когда компилятор увидит вызов такой функции, он сгенерирует код для функции, поставив в неё нужный тип данных. Компилятор принимает решение о том, как именно компилировать функцию, основываясь только на типе данных используемого в шаблоне аргумента. Тип данных, возвращаемый функцией, не играет при этом роли.

В шаблоне функции можно использовать несколько шаблонных аргументов. Шаблонные функции можно перегружать.

Когда компилятор встречает вызов какой-то функции, для его разрешения он следует такому алгоритму

- Сначала ищется обычная функция с соответствующими параметрами;

- Если таковой не найдено, компилятор ищет шаблон, из которого можно было бы генерировать функцию с точным соответствием параметров;
- Если этого сделать невозможно, компилятор вновь рассматривает обычные функции на предмет возможных преобразований типа параметров

ШАБЛОНЫ КЛАССОВ

Шаблонный принцип можно расширить и на классы. В этом случае шаблоны используются, когда класс является хранилищем данных.

Пусть, например, имеется класс типа стек, для хранения чисел типа int.

class Stack

{

private:

int st[max]; // целочисленныймассив

int top; // индекс вершины стека

public:

Stack( ); // конструктор

voidpush(intvar); // аргумент типа int

int pop( ); // возвращает значение типа int

};

Если теперь нужно будет хранить в стеке значения типа float, придется написать новый класс.

class Stack

{

private:

float st[max]; // массив

int top; // индексвершиныстека

public:

Stack( ); // конструктор

voidpush(floatvar); // аргумент типа float

float pop( ); // возвращает значение типа float

};

Подобным же образом пришлось бы создать классы для хранения данных каждого типа. Для преодоления этого ограничения и используются шаблоны классов.

```
template <class Type>

class Stack

{

private:

Type st[max]; // массивлюбоготипа

int top; // индексвершиныстека

public:

Stack( ); // конструктор

void push(Type var); // аргументлюбоготипа

Type pop( ); // возвращает значение любого типа

};
```

Здесь *Stack* является шаблонным классом. Идея шаблонных классов во многом сходна с идеей шаблонных функций. Шаблоны классов отличаются от шаблонов функций способом реализации. Для создания шаблонной функции она вызывается с аргументами нужного типа. Классы реализуются с помощью определения объекта, использующего шаблонный аргумент.

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

Такое выражение создаст переменную s1. В нашем случае это будет стек, в котором хранятся числа типа float. На рисунке показано, как шаблоны классов и определения конкретных объектов приводят к занесению этих объектов в память.

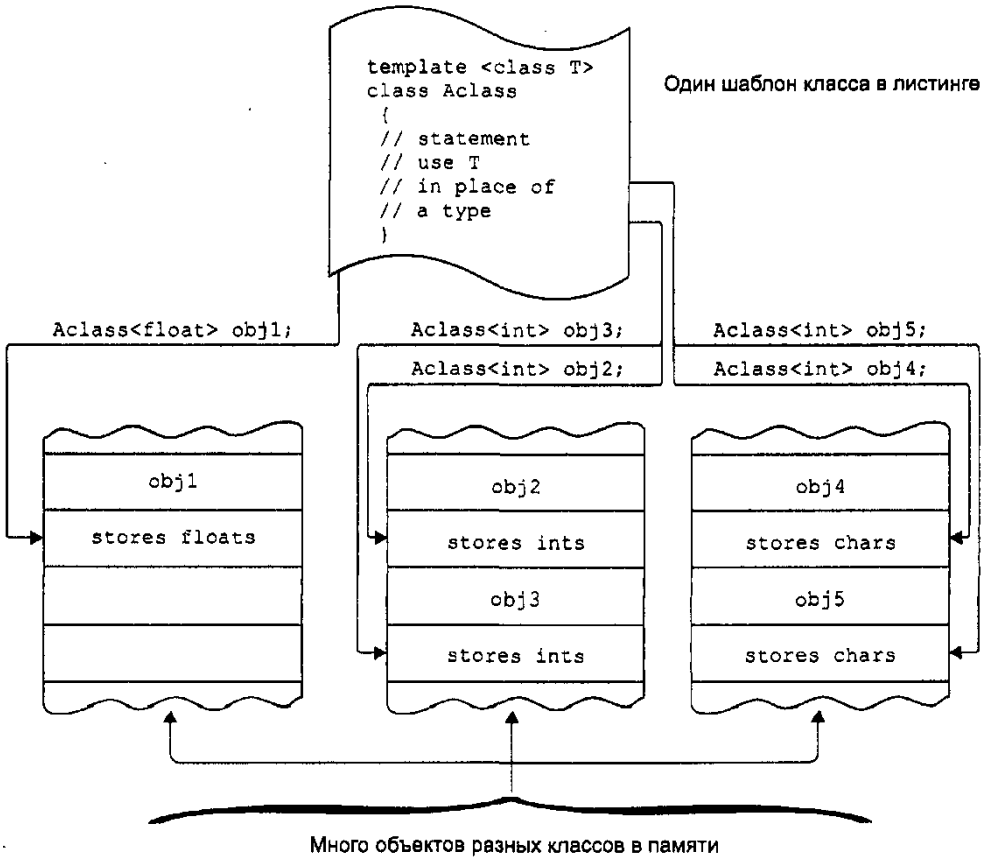

Рисунок 2. Шаблон класса.

Если методы класса определяются вне его спецификации, то они становятся, по сути, шаблонами функций, и определять их надо именно так. Например, конструктор класса стек из предыдущего примера будет описан так:

template<class Type>

Stack<Type>::Stack( )

{

top = -1;

}

то есть выражение *template <class Type>* должно предварять не только определение класса, но и каждый определённый вне класса метод.

В библиотеке C++ Standard Template Library реализовано множество шаблонов стандартных типов данных (списки, векторы, очереди, реализованные как шаблоны классов) и стандартных методов (накопление, поиск и сортировка и др.). Подробнее узнать о них можно из литературы.

УПРАВЛЕНИЕ ИСКЛЮЧЕНИЯМИ

Исключения позволяют применить объектно-ориентированный подход к обработке возникающих в классах ошибок. Под исключениями понимаются ошибки, возникающие во время работы программы. Они могут быть вызваны различными обстоятельствами,

такими как выход за пределы массива, ошибка открытия файла, инициализация объекта некорректным значением и т.д.

В нормальной ситуации вызовы методов классов не приводят ни к каким ошибкам. Но иногда в программе возникает ошибка, которую обнаруживает *сам метод*. Например, это может быть проверка на выход за пределы индексов массива. И тогда метод информирует программу о случившемся – генерирует исключительную ситуацию. В приложении при этом создается отдельная секция кода, в которой задаются операции по обработке ошибок - этот блок называют обработчиком исключительных ситуаций (см рисунок).

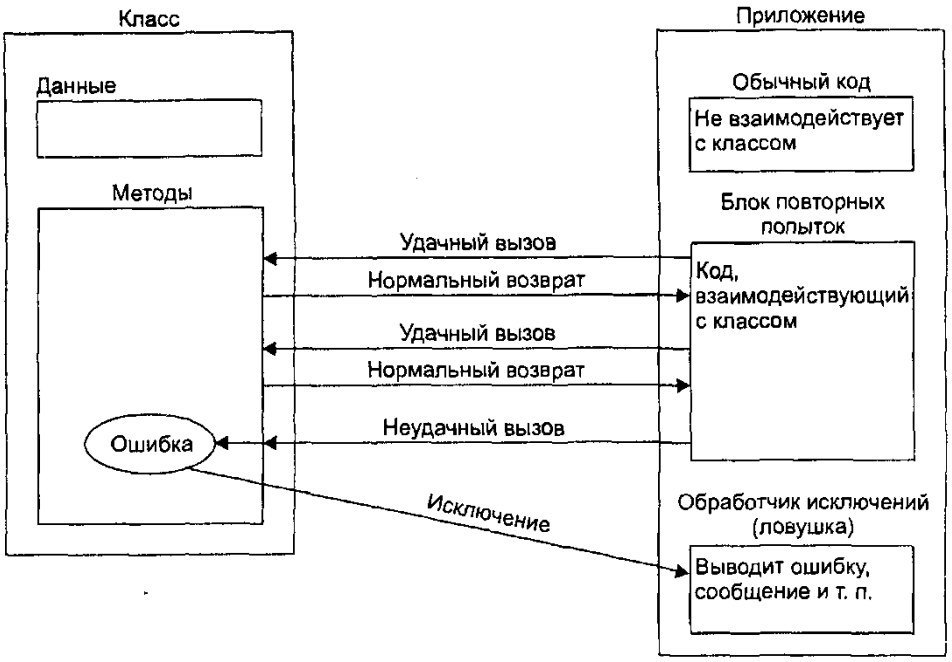

Рисунок 3. Механизм исключений.

Механизм исключений использует три служебных слова: catch, throw и try. Ниже приводится пример программы, демонстрирующий механизм исключений.

```
#include <iostream>

using namespace std;

const int MAX = 3; //в стеке максимум 3 целых числа

//////////////////////////////////////////////////////////

class Stack

{

private:

int st[MAX]; //стек: целочисленныймассив

int top; //индексвершиныстека

public:
```

```
classRange //классисключенийдляStack
{ //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! Стекпуст

}

catch(Stack::Range) //обработчик

{

cout << "Исключение: Стек переполнен или пуст"<<endl;

}

cout << "Приехали сюда после захвата исключения (или нормального выхода" << endl;

return 0;

}

В приведенном примере класс исключение описывается внутри класса *Stack* следующим образом

class Range

{

};

Тело класса пусто, в данном случае он создается исключительно ради имени класса. Оно используется для связывания выражения генерации исключения *throw* с улавливающим блоком *catch*.

В классе *Stack* исключение возникает, когда приложение пытается извлечь значение из пустого стека, или положить значение в уже заполненный стек. Чтобы сообщить приложению о том, что оно выполнило недопустимую операцию с объектом, методы этого класса проверяют условия с использованием if и генерируют исключение, если условие выполняется. В примере исключение генерируется в двух местах, с помощью выражения

throw Range( );

Все выражения, в который могут произойти ошибки, заключены в фигурные скобки, перед которыми стоит слово *try*. Этот блок называется блоком повторных попыток.

Код, в котором содержатся операции по обработке ошибок, заключается в фигурные скобки и начинается со слова catch. В скобках указывается имя класса обрабатываемого исключения. В примере это *catch(Stack::Range)*. Если класс ошибки не важен (то есть нужно обработать любую ошибку), то в скобках указываются три точки *catch(...)*.

Классы исключений не обязательно объявлять внутри класса, они могут и не принадлежать другим классам. Можно в качестве классов исключений использовать и встроенные типы данных.

Можно спроектировать класс и таким образом, чтобы он генерировал несколько исключений. В приведенном ниже примере описан класс стека, который генерирует разные исключения для ситуаций пустого и заполненного стека.

```
#include <iostream>

usingnamespacestd;

constintMAX = 3; //встекеможетбытьдотрехцелыхчисел

//////////////////////////////////////////////////////////

class Stack

{

private:

int st[MAX]; //стек: массивцелыхчисел

int top; //индексверхушкистека

public:

class Full { }; //классисключения

classEmpty { }; //классисключения

//--------------------------------------------------------

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(); //генерироватьисключение 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: стекпуст
}
catch(Stack::Full)
{
cout<< "Ошибка: переполнениестека" <<endl;
}
```

```
catch(Stack::Empty)

{

cout<< "Ошибка: стекпуст" <<endl;

}

return 0;

}
```

Существуют и более сложные конструкции использования исключений. Такие как, например, исключения с аргументами. Они предназначены для передачи в программу дополнительных сведений о том, что привело к возникновению исключительной ситуации. Познакомиться с ними можно, обратившись к специальной литературе.