Глава 11

Шаблоны и обработка исключительных ситуаций

В этой главе вы познакомитесь с двумя важнейшими характеристиками C++ верхнего уровня: *шаблонами (templates)* и *обработкой исключительных ситуаций (exception handling)*. Ни та, ни другая характеристики не входили в изначальную спецификацию C++, а были добавлены туда несколько лет назад и определены в стандарте Standard C++. Эти характеристики поддерживаются всеми современными компиляторами и позволяют достичь двух наиболее заманчивых целей программирования: создания многократно используемых и отказоустойчивых программ.

С помощью шаблонов можно создавать родовые функции (generic functions) и родовые классы (generic classes). В родовой функции или классе тип данных, с которыми функция или класс работают, задается в качестве параметра. Это позволяет одну и ту же функцию или класс использовать с несколькими различными типами данных без необходимости программировать новую версию функции или класса для каждого конкретного типа данных. Таким образом шаблоны дают возможность создавать многократно используемые программы. В данной главе рассказывается как о родовых функциях,
так и о родовых классах.

Система обработки исключительных ситуаций встроена в C++ и позволяет работать с ошибками, которые возникают во время работы программы, заранее предусмотренным и управляемым образом. С помощью системы обработки исключительных ситуаций C++ ваша программа при возникновении ошибки может автоматически вызвать процедуру ее обработки. Принципиальное преимущество обработки исключительных ситуаций состоит в том, что она автоматически в зависимости от ситуации запускает одну из множества подпрограмм обработки ошибок, которые предварительно "вручную" встраиваются в основную программу. Должным образом запрограммированная обработка исключительных ситуаций помогает создавать действительно отказоустойчивые программы.

11.1. Родовые функции

Родовая функция определяет базовый набор операций, которые будут применяться к разным типам данных. Родовая функция оперирует с тем типом данных, который она получает в качестве параметра. С помощью этого механизма одна и та же процедура может применяться к самым разным данным. Как известно, многие алгоритмы логически одинаковы, независимо от того, для обработки каких типов данных они предназначены. Например, алгоритм быстрой сортировки одинаков как для массивов целых, так и для массивов действительных чисел. Это именно тот случай, когда сортируемые данные отличаются только по типам. Благодаря созданию родовой функции вы можете независимо от типа данных определить суть алгоритма. После того как это сделано, компилятор автоматически генерирует правильный код для фактически используемого при выполнении функции типа данных. По существу, при создании родовой функции вы создаете функцию, которая может автоматически перегружаться сама.

Родовая функция создается с помощью ключевого слова template. Обычное значение этого слова (т. е. шаблон) точно отражает его назначение в C++. Оно предназначено для создания шаблона (или каркаса), который описывает то, что будет делать функция, при этом

компилятору остается дополнить <каркас необходимыми деталями. Ниже представлена типовая форма определения функции-шаблона:

template <class *Фтип*> *возвр_значение имя_функции*(*список_параметров*)
{
// тело функции
}

Здесь вместо ***Фтип*** указывается тип используемых функцией данных. Это имя можно указывать внутри определения функции. Однако это всего лишь фиктивное имя, которое компилятор автоматически заменит именем реального типа данных при создании конкретной версии функции.

Хотя традиционно для задания родового типа данных в объявлении шаблона указывается ключевое слово **class**, вы также можете использовать ключевое слово **typename**.

Примеры

1. В следующей программе создается родовая функция, которая меняет местами значения двух переменных, передаваемых ей в качестве параметров. Поскольку в своей основе процесс обмена двух значений не зависит от типа переменных, этот процесс удачно реализуется с помощью родовой функции.

```
// Пример родовой функции или шаблона
#include <iostream>
using namespace std;

// Это функция-шаблон
template <class X> void swapargs(X &a, X &b)
{
   X temp;

   temp = a;
   a = b;
   b = temp;
}

int main()
{
   int i = 10, j = 20;
   float x = 10.1, y = 23.3;

   cout << "Исходные значения i, j равны: " << i << ' ' << j << endl;
   cout << "Исходные значения x, y равны: " << x << ' ' << y << endl;

   swapargs(i, j);    // обмен целых
   swap(x, y);        // обмен действительных

   cout << "Новые значения i, j равны: " << i << ' ' << j << endl;
   cout << "Новые значения x, y равны: " << x << ' ' << y << endl;

   return 0;
}
```

Ключевое слово **template** используется для определения родовой функции. Строка:

```
template <class X> void swapargs(X &a, X &b)
```

сообщает компилятору две вещи: во-первых, создается шаблон, и, во-вторых, начинается определение родовой функции. Здесь **X** - это родовой тип данных, используемый в качестве фиктивного имени. После строки с ключевым словом **template** функция **swapargs()** объявляется с именем **X** в качестве типа данных обмениваемых значений. В функции **main()** функция **swapargs()** вызывается с двумя разными типами данных: целыми и действительными. Поскольку функция **swapargs()** - это родовая функция, компилятор автоматически создает две ее версии: одну - для обмена целых значений, другую для обмена действительных значений. Теперь попытайтесь скомпилировать программу.

Имеются и другие термины, которые можно встретить при описании шаблонов в литературе по C++. Во-первых, родовая функция (то есть функция, в определении которой имеется ключевое слово **template**) также называется *функция-шаблон (template function).* Когда компилятор создает конкретную версию этой функции, говорят, что он создал *порожденную функцию (generated function).* Процесс генерации порожденной функции называют *созиданием экземпляра (instantiating)* функции. Другими словами, порожденная функция-это конкретный экземпляр функции-шаблона. I

2. Ключевое слово **template** в определении родовой функции не обязательно должно быть в той же строке, что и имя функции. Например, ниже приведен еще один вполне обычный формат определения функции **swapargs()**:

```
template <class X>
void swapargs(X &a, X &b)
{
X temp;
temp = a;
a = b;
b = temp;
}
```

При использовании такого формата важно понимать, что никаких других инструкций между инструкцией **template** и началом определения родовой функции быть не может. Например, следующий фрагмент программы компилироваться не будет:

```
// Этот фрагмент компилироваться не будет
template <class X>
int i;              // это неправильно
void swapargs(X &a, X &b)
{
X temp;
temp = a;
a = b;
b = temp;
}
```

Как указано в комментариях, инструкция с ключевым словом **template** должна находиться сразу перед определением функции.

3. Как уже упоминалось, для задания родового типа данных в определении шаблона вместо ключевого слова **class** можно указывать ключевое слово **typename**. Например, ниже приведено еще одно объявление функции
swapargs():

```
// Использование ключевого слова typename
template <typename X> void swapargs(X &a, X &b)
```

```
{
X temp;
temp = a;
a = b;
b = temp;
}
```

Ключевое слово **typename** можно также указывать для задания неизвестного типа данных внутри шаблона, но такое его использование выходит за рамки данной книги.

4. С помощью инструкции **template** можно определить более одного родового типа данных, отделяя их друг от друга запятыми. Например, в данной программе создается родовая функция, в которой имеются два родовых типа данных:

```
#include <iostream>
using namespace std;
template <class typel, class type2>
void myfunc(typel x, type2 y)
{
cout < x < ' ' < y < endl;
}

int main()
{
myfunc(10, "hi");
myfunc(0.23, 10L); '
return 0 ;
}
```

В данном примере при генерации конкретных экземпляров функции **myfunc()**, фиктивные имена типов **typel** и **type2** заменяются компилятором на типы данных **int** и **char*** или **double** и **long** соответственно.

5. Родовые функции похожи на перегруженные функции за исключением того, что они более ограничены по своим возможностям. При перегрузке функции внутри её тела можно выполнять совершенно разные действия. С другой стороны, родовая функция должна выполнять одни и те же базовые действия для всех своих версий. Например, следующие перегруженные функции *нельзя* заменить на родовую функцию, поскольку они делают не одно и то же.

```
void outdata (int i)
{
cout << i;
}
void outdata (double d)
{
cout << setpresicion(10) << setfill('#');
cout << d;
cout << setpresicion(6) << setfill(' ');
}
```

6. Несмотря на то, что функция-шаблон при необходимости перегружается сама, её также можно перегрузить явно. Если вы сами перегружаете родовую функцию, то перегруженная функция подменяет (или «скрывает») родовую функцию, которую бы создал компилятор для этой конкретной версии. Рассмотрим такой вариант примера 1:

```
// Подмена функции-шаблона
#include <iostream>
```

```
using namespace std;
template <class X> void swapargs(X &a, X &b)
{
   X temp;
   temp = a;
   a = b;
   b = temp;
}

// Здесь переопределяется родовая версия функции swapargs()
void swapargs(int a, int b)
{
   cout << "это печатается внутри функции swapargs(int, int)\n";
}

int main()
{
   int i = 10, j = 20;
   float x = 10.1, y = 23.3;
   cout << "Исходные значения i, j равны: " << i << ' ' << j << endl;
   cout << "Исходные значения x, y равны: " << x << ' ' << y << endl;
   swapargs(i, j);    // вызов явно перегруженной функции swapargs()
   swapargs(x, y);    // обмен действительными числами
   cout << "Новые значения i, j равны: " << i << ' ' << j << endl;
   cout << "Новые значения x, y равны: " << x << ' ' << y << endl;
   return 0;
}
```

Как отмечено в комментариях, при вызове функции **swapargs(i,j)** вызывается определённая в программе явно перегруженная версия функции **swapargs().** Таким образом, компилятор не генерирует этой версии родовой функции **swapargs()**, поскольку родовая функция подменяется явно перегруженной функцией.

Ручная перегрузка шаблона, как показано в данном примере, позволяет изменить версию родовой функции так, чтобы приспособить её к конкретной ситуации. Однако в большинстве случаев, если вам нужно несколько разных версий функции для разных типов данных, вместо шаблонов лучше использовать перегруженные функции.

11.2. Родовые классы

В дополнение к родовым функциям можно определить и родовые классы. При этом создается класс, в котором определены все необходимые алгоритмы, а фактические типы обрабатываемых данных задаются в качестве параметров позже, при создании объектов этого класса.

Родовые классы полезны, когда класс содержит общую логику работы. Например, алгоритм, который реализует очередь целых, будет также работать и с очередью символов.. Кроме того, механизм, который реализует связанный список почтовых адресов, будет также поддерживать связанный список запасных частей к автомобилям. С помощью родового класса можно создать класс, реализующий очередь, связанный список и т. д. для любых типов данных. Компилятор будет автоматически генерировать правильный тип объекта на основе типа, заданного при создании объекта.

Ниже представлена основная форма объявления родового класса:

template <class *Фтип*> class *имя_класса* {
.
.
.
}

Здесь ***Фтип*** - это фиктивное имя типа, который будет задан при создании экземпляра класса. При необходимости можно определить более одного родового типа данных, разделяя их запятыми.

После создания родового класса с помощью представленной ниже форме можно создать конкретный экземпляр этого класса:

имя_класса <тип> объект;

Здесь ***тип*** - это имя типа данных, с которым будет оперировать класс.

Функции-члены родового класса сами автоматически становятся родовыми. Для них не обязательно явно задавать ключевое слово **template.**

Как вы увидите в главе 14, в C++ имеется встроенная библиотека классов-шаблонов, которая называется библиотекой стандартных шаблонов (Standard Template Library, STL). Эта библиотека предоставляет родовые версии классов для наиболее часто используемых алгоритмов и структур данных. Чтобы научиться пользоваться библиотекой стандартных шаблонов с максимальной эффективностью, вам необходимо иметь твердые знания по классам-шаблонам и их синтаксису.

Примеры

1. В следующей программе создается очень простой родовой класс, реализующий односвязный список. Затем демонстрируются возможности такого класса путем создания связанного списка для хранения символов.

```
// Простой родовой связанный список
#include <iostream>
using namespace std;

template <class data_t> class list {
   data_t data;
   list *next;
public:
   list (data_t d);
   void add(list *node) { node->next = this; next = 0; }
   list *getnext() { return next; }
   data_t getdata() { return data; }
}

template <class data_t> list<data_t>::list(data_t d)
{
   data = d;
   next = 0;
}

int main()
{
```

```
    list<char> start('a');
    list<char> *p, *last;
    int i;
    // создание списка
    last = &start;
    for(i=1; i<26; i++) {
      p = new list<char> ('a' + i);
      p->add(last);
      last = p;
    }
    // вывод списка
    p = &start;
    while(p) {
      cout << p->getdata();
      p = p->getnext();
    }
    return 0;
}
```

Как видите, объявление родового класса похоже на объявление родовой функции. Тип данных, хранящихся в списке, становится родовым в объявлении класса. Но он не проявляется, пока не объявлен объект, который и задает реальный тип данных. В данном примере объекты и указатели создаются внутри функции **main(),** где указывается, что типом хранящихся в списке данных является тип **char**. Обратите особое внимание на следующее объявление:

```
list<char> start('a');
```

Отметьте, что необходимый тип данных задаётся между угловыми скобками.

Наберите и выполните эту программу. В ней создаётся связанный список с символами алфавита, который затем выводится на экран. Путём простого изменения типа данных, который указывается при создании объектов, можно изменить тип данных, хранящихся в списке. Например, с помощью следующего объявления можно создать другой объект, где можно было бы хранить целые:

```
list<int> int_start(1);
```

Можно также использовать список **list** для хранения создаваемых вами типов данных. Например, для хранения адресной информации можно воспользоваться следующей структурой:

```
struct addr {
        char name[40];
        char street[40];
        char city[30];
        char state[3];
        char zip[12];
}
```

Теперь, чтобы с помощью списка **list** хранить объекты типа **addr**, используйте такое объявление (предположим, что объект типа **structvar** содержит правильную структуру **addr**):

```
list<addr> obj(structvar);
```

2. Ниже представлен другой пример родового класса. Это переработанный класс **stack**, впервые приведенный в главе 1. Однако в данном случае класс **stack** реализован как шаблон. Следовательно, в нём можно хранить объекты любого типа. В представленном ниже примере создаются стек символов и стек действительных чисел:

```
// Здесь показан родовой стек
#include <iostream>
using namespace std;

#define SIZE 10

// Создание родового класса stack
template <class StackType> class stack {
   StackType stck[SIZE]; // содержит стек
   int tos;              // индекс вершины стека

public:
   void init() { tos = 0; } // инициализация стека
   void push(StackType ch); // помещает объект в стек
   StackType pop();         // выталкивает объект из стека
};

// Помещение объекта в стек
template <class StackType>
void stack<StackType>::push(StackType ob)
{
   if (tos==SIZE) {
      cout << "Стек полон";
      return;
   }
   stck[tos] = ob;
   tos++;
}

// Выталкивание объекта из стека
template <class StackType>
StackType stack<StackType>::pop()
{
   if (tos==0) {
      cout << "Стек пуст";
      return 0; // возврат нуля при пустом стеке
   }
   tos--;
   return stck[tos];
}

int main()
{
   // Демонстрация символьных стеков
   stack<char> s1, s2; // создание двух стеков
   int i;

   // инициализация стеков
   s1.init();
   s2.init();
   s1.push('a');
   s2.push('x');
   s1.push('b');
   s2.push('y');
   s1.push('c');
   s2.push('z');
```

```
  for(i=0; i<3; i++) cout << "Из стека 1:" << s1.pop() << "\n";
  for(i=0; i<3; i++) cout << "Из стека 2:" << s2.pop() << "\n";

  // Демонстрация стеков со значениями типа double
  stack<double> ds1, ds2; // создание двух стеков

  // инициализация стеков
  ds1.init();
  ds2.init();
  ds1.push(1.1);
  ds2.push(2.2);
  ds1.push(3.3);
  ds2.push(4.4);
  ds1.push(5.5);
  ds2.push(6.6);
  for(i=0; i<3; i++) cout << " Из стека 1:" << ds1.pop() << "\n";
  for(i=0; i<3; i++) cout << " Из стека 2:" << ds2.pop() << "\n";
  return 0;
}
```

Как показано на примере класса **stack** (и предыдущего класса **list**), родовые функции и классы обеспечивают мощный инструмент экономии времени при программировании, поскольку они позволяют определить общую форму алгоритма, который затем можно использовать с данными любого типа. Таким образом, вы избегаете однообразных операций по созданию отдельных процедур для каждого типа данных, с которыми должен работать ваш алгоритм.

3. Класс-шаблон может иметь более одного родового типа данных. Просто объявите все необходимые для класса типы данных внутри спецификации **template**, перечислив их через запятую. Например, в следующем коротком примере создается класс с двумя родовыми типами данных:

```
// Здесь в определении класса используется два родовых типа данных
#include <iostream>
using namespace std;

template <class Type1, class Type2> class myclass
{
  Type1 i;
  Type2 j;
public:
  myclass(Type1 a, Type2 b) { i = a; j = b; }
  void show() { cout << i << ' ' << j << '\n'; }
};

int main()
{
  myclass<int, double> ob1(10, 0.23);
  myclass<char, char *> ob2('X', "Это проверка");
  ob1.show();  // вывод значений типа int и double
  ob2.show();  // вывод значений типа char и char *
}
```

После выполнения программы на экран выводится следующая информация:

```
10 0.23
X Это проверка
```

В программе объявлено два типа объектов. В объекте **ob1** используются целое и значение двойной точности. В объекте **ob2** - символ и указатель на символ. В обоих случаях компилятор автоматически генерирует необходимые данные и функции в соответствии со способом создания объектов.

11.3. Обработка исключительных ситуаций

C++ обеспечивает встроенный механизм обработки ошибок, называемый *обработкой исключительных ситуаций (exception handling).* Благодаря обработке исключительных ситуаций можно упростить управление и реакцию на ошибки во время выполнения программ. Обработка исключительных ситуаций в C++ организуется с помощью трех ключевых слов: **try**, **catch** и **throw**.

В самых общих словах, инструкции программы, во время выполнения которых вы хотите обеспечить обработку исключительных ситуаций, располагаются в блоке **try**. Если исключительная ситуация (т. е. ошибка) имеет место внутри блока **try**, она возбуждается (ключевое слово **throw**), перехватывается (ключевое слово **catch**) и обрабатывается. Ниже поясняется приведенное здесь общее описание.

Как уже отмечалось, любая инструкция, которая возбуждает исключительную ситуацию, должна выполняться внутри блока **try**. (Функции, которые вызываются из блока **try** также могут возбуждать исключительную ситуацию.) Любая исключительная ситуация должна перехватываться инструкцией **catch**, которая располагается непосредственно за блоком **try**, возбуждающем исключительную ситуацию. Далее представлена основная форма инструкций **try** и **catch**:

```
try {
// блок возбуждения исключительной ситуации
}

catch (typel arg) {
// блок перехвата исключительной ситуации
}

catch (type2 arg) {
// блок перехвата исключительной ситуации
}

catch (type3 arg) {
// блок перехвата исключительной – ситуации
}
.
.
.
catch (typeN arg) {
// блок перехвата исключительной ситуации
}
```

Блок **try** должен содержать ту часть вашей программы, в который вы хотите отслеживать ошибки. Это могут быть как несколько инструкций внутри одной функции, так и все

инструкции внутри функции **main()** (что естественно ведет к отслеживанию ошибок во всей программе).

После того как исключительная; ситуация возбуждена, она перехватывается соответствующей этой конкретной исключительной ситуации инструкцией **catch**, которая ее обрабатывает. С блоком **try** может быть связано более одной инструкции catch. То, какая именно инструкция **catch** используется, зависит от типа исключительной ситуации. То есть, если тип данных, указанный в инструкции **catch**, соответствует типу исключительной ситуации, выполняется данная инструкция **catch**. При этом все оставшиеся инструкции блока **try** игнорируются (т. е. сразу после того, как какая-то инструкция в блоке **try** вызвала появление исключительной ситуации, управление передается соответствующей инструкции **catch**, минуя оставшиеся инструкции блока **try**, - примеч. пер.). Если исключительная ситуация перехвачена, аргумент ***arg*** получает ее значение. Если вам не нужен доступ к самой исключительной ситуации, можно в инструкции **catch** указать только ее тип ***type***, аргумент ***arg*** указывать не обязательно. Можно перехватывать любые типы данных, включая и типы создаваемых вами классов. Фактически в качестве исключительных ситуаций часто используются именно типы классов.

Далее представлена основная форма инструкции throw:

throw ***исключительная_ситуация*****;**

Инструкция throw должна выполняться либо внутри блока **try**, либо в любой функции, которую этот блок вызывает (прямо или косвенно). Здесь ***исключительная_ситуация*** - это возбуждаемая инструкцией **throw** исключительная ситуация.

Если вы возбуждаете исключительную ситуацию, для которой нет соответствующей инструкции **catch**, может произойти ненормальное завершение программы. Если ваш компилятор функционирует в соответствии со стандартом Standard C++, то возбуждение необрабатываемой исключительной ситуации приводит к вызову стандартной библиотечной функции **terminate().** По умолчанию для завершения программы функция **terminate()** вызывает функцию **abort()** однако при желании можно задать собственную процедуру завершения программы. За подробностями обращайтесь к справочной документации вашего компилятора.

Примеры

1. В следующем очень простом примере показано, как в C++ функционирует система обработки исключительных ситуаций:

```
// Простой пример обработки исключительной ситуации
#include <iostream>
using namespace std;

int main()
{
   cout << "начало\n";
   try {          // начало блока try
      cout << "Внутри блока try\n";
      throw 10;   // возбуждение ошибки
      cout << "Эта инструкция выполнена не будет";
   }
   catch (int i) { // перехват ошибки
      cout << "перехвачена ошибка номер: ";
      cout << i << "\n";
```

```
    }
    cout << "конец";
    return 0;
}
```

После выполнения программы на экране появится следующее:

```
начало
Внутри блока try
Перехвачена ошибка номер: 10
конец
```

Внимательно изучите программу. Как видите, здесь имеется блок **try**, содержащий три инструкции, и инструкция **catch (int i)**, обрабатывающая исключительную ситуацию целого типа. Внутри блока **try** будут выполнены только две из трёх инструкций – инструкция **cout** и **throw**. После того как исключительная ситуация возбуждена, управление передаётся выражению **catch** и выполнение блока **try** завершается. Таким образом, инструкция **catch** вызывается *не* явно, управление выполнением программы просто передаётся этой инструкции. (Для этого стек автоматически сбрасывается.) Следовательно, следующая за инструкцией **throw** инструкция **cout** не будет выполнена никогда.

После выполнения инструкции **catch**. Управление программы передаётся следующей за ней инструкции. Тем не менее, обычно блок **catch** заканчивают вызовом функции **exit(), abort()** или какой-либо другой функции, принудительно завершающей программу, поскольку, как правило, система обработки исключительных ситуаций предназначена для обработки катастрофических ошибок.

2. Как уже упоминалось, тип исключительной ситуации должен соответствовать типу, заданному в инструкции **catch**. Например, в предыдущем примере, если изменить тип данных в инструкции **catch** на **double**, то исключительная ситуация не будет перехвачена и будет иметь место ненормальное завершение программы. Это продемонстрировано в следующем фрагменте:

```
// Этот пример работать не будет
#include <iostream>
using namespace std;

int main()
{
    cout << "начало\n";
    try {             // начало блока try
        cout << "Внутри блока try\n";
        throw 10;      // возбуждение ошибки
        cout << "Эта инструкция выполнена не будет";
    }
    catch (double i) { // Эта инструкция не будет работать
                       // с исключительной ситуацией целого типа
        cout << "перехвачена ошибка номер: ";
        cout << i << "\n";
    }
    cout << "конец";
    return 0;
}
```

Поскольку исключительная ситуация целого типа не будет перехвачена инструкцией **catch** типа **double**, на экран программа выведет следующее:

Начало
Внутри блока **try**
Abnormal program termination

3. Исключительная ситуация может быть возбуждена не входящей в блок **try** инструкцией, если сама эта инструкция входит в эту функцию, которая вызывается из блока **try**. Например, ниже представлена совершенно правильная программа:

```
/* Возбуждение исключительной ситуации из функции, находящейся вне блока try
*/
#include <iostream>
using namespace std;

void Xtest(int test)
{
  cout << "Внутри функции Xtest, test равно: " << test << "\n";
  if(test) throw test;
}

int main()
{
  cout << "начало\n";
  try {          // начало блока try
    cout << "Внутри блока try\n";
    Xtest(0);
    Xtest(1);
    Xtest(2);
  }
  catch (int i) { // перехват ошибки
    cout << "перехвачена ошибка номер: ";
    cout << i << "\n";
  }
  cout << "конец";
  return 0;
}
```

На экран программа выводит следующее:

```
начало
Внутри блока try
Внутри функции Xtest, test равно: 0
Внутри функции Xtest, test равно: 1
Перехвачена ошибка номер: 1
конец
```

4. В блоке **try** можно располагать внутри функции. В этом случае при каждом входе в функцию обработчик исключительной ситуации устанавливается снова. Например, рассмотрим следующую программу:

```
#include <iostream>
using namespace std;

// Блоки try и catch могут находиться не только в функции main()
void Xhandler(int test)
{
  try {
    if(test) throw test;
  }
  catch(int i) {
    cout << "перехвачена ошибка номер: " << i << '\n';
```

```
    }
}

int main()
{
    cout << "начало\n";
    Xhandler(1);
    Xhandler(2);
    Xhandler(0);
    Xhandler(3);
    cout << "конец";
    return 0;
}
```

На экран программа выводит следующее:

```
начало
Перехвачена ошибка номер: 1
Перехвачена ошибка номер: 2
Перехвачена ошибка номер: 3
конец
```

Как видите, обработаны три исключительные ситуации. После вызова каждой исключительной ситуации функция возвращает своё значение. При повторном вызове функции обработчик исключительной ситуации устанавливается вновь.

5. Как упоминалось ранее, с блоком **try** можно связать более одной инструкции **catch**. Как правило, так и делается. При этом каждая инструкция **catch** предназначена для перехвата своего типа исключительной ситуации. Например, в следующей программе перехватываются две исключительных ситуации, одна для целых и одна для строки:

```
#include <iostream>
using namespace std;

// Можно перехватывать разные типы исключительных ситуаций
void Xhandler(int test)
{
    try {
        if(test) throw test;
        else throw "Значение равно нулю";
    }
    catch(int i) {
        cout << "Перехвачена ошибка номер: " << i << '\n';
    }
    catch(char *str) {
        cout << "Перехвачена стока: ";
        cout << str << '\n';
    }
}
int main()
{
    cout << "начало\n";
    Xhandler(1);
    Xhandler(2);
    Xhandler(0);
    Xhandler(3);
    cout << "конец";

    return 0;
}
```

На экран программа выводит следующее:

```
начало
Перехвачена ошибка номер: 1
Перехвачена ошибка номер: 2
Перехвачена строка: Значение равно нулю
Перехвачена ошибка номер: 3
Конец
```

Как видите, каждая инструкция **catch** перехватывает только исключительные ситуации соответствующего ей типа.

Обычно выражения инструкций **catch** проверяются в том порядке, в котором они появляются в программе. Выполняется только та инструкция, которая совпадает по типу данных с исключительной ситуацией. Все остальные блоки **catch** игнорируются.

11.4. Дополнительная информация об обработке исключительных ситуаций

В системе обработки исключительных ситуаций имеется несколько дополнительных аспектов и нюансов, которые могут сделать ее понятнее и удобней для применения.

В некоторых случаях необходимо настроить систему так, чтобы перехватывать все исключительные ситуации, независимо от их типа. Сделать это достаточно просто. Для этого используйте следующую форму инструкции **catch**:

```
catch (...) {
// обработка всех исключительных ситуаций
}
```

Здесь многоточие соответствует любому типу данных.

Для функции, вызываемой из блока **try**, вы можете ограничить число типов исключительных ситуаций, которые способна возбудить такая функция. Фактически можно даже запретить функции вообще возбуждать какую бы то ни было исключительную ситуацию. Для этого необходимо добавить в определение функции ключевое слово **throw**. Здесь представлена основная форма такого определения:

возвращаемый_тип имя_функции (список_аргументов) **throw** ***(список_типов)***
```
{
// ...
}
```

Здесь в поле ***список_типов*** перечисляются через запятые только те типы данных исключительных ситуаций, которые могут быть возбуждены функцией. Возбуждение любого другого типа исключительной ситуации приведет к аварийному завершению программы. Если вы хотите, чтобы функция не возбуждала никаких исключительных ситуаций, то оставьте поле ***список_типов*** пустым.

Если ваш компилятор работает в соответствии с современным стандартом Standard C++, то попытка возбуждения неподдерживаемой исключительной ситуации приведет к вызову

стандартной библиотечной функции **unexpected().** По умолчанию функция **unexpected()** вызывает функцию **abort(),** что ведет к аварийному завершению программы. Однако при желании вы можете задать собственную процедуру завершения программы. За подробностями обращайтесь к справочной документации вашего компилятора.

Если вы хотите в процедуре обработки исключительной ситуации возбудить повторную исключительную ситуацию, можно просто использовать инструкцию **throw** без параметров. Это приводит к тому, что текущая исключительная ситуация передается внешней последовательности инструкций **try/catch**.

Примеры

1. В следующей программе иллюстрируется использование инструкции **catch(...):**

```
// В этой программе перехватываются все типы исключительных ситуаций
#include <iostream>
using namespace std;

void Xhandler(int test)
{
   try {
      // возбуждение исключительной ситуации типа int
      if(test==0) throw test;
      // возбуждение исключительной ситуации типа char
      if(test==1) throw 'a';
      // возбуждение исключительной ситуации типа double
      if(test==2) throw 123.23;
   }
   catch(...) { // перехват исключительных ситуаций всех типов
      cout << "Перехвачена ошибка!\n";
   }
}

int main()
{
   cout << "начало\n";
   Xhandler(0);
   Xhandler(1);
   Xhandler(2);
   cout << "конец";
   return 0;
}
```

На экран программа выводит следующее:

```
начало
Перехвачена ошибка!
Перехвачена ошибка!
Перехвачена ошибка!
конец
```

Как видите, во всех трёх случаях возбуждения исключительной ситуации в инструкции **throw**, она перехватывается с помощью единственной инструкции **catch**.

2. Очень удобно инструкцию **catch(…)** использовать в качестве последней в группе инструкций **catch**. В єтом случае инструкция **catch(…)** по умолчанию становится инструкцией, которая “перехватывает всё”. Например, далее представлена слегка

изменённая версия предыдущей программы, где исключительные ситуации целого типа перехватываются явно, а другие – с помощью инструкции **catch(…)** :

```
/* В этом примере инструкция catch(...) по умолчанию перехватывает все типы исключительных ситуаций
*/
#include <iostream>
using namespace std;

void Xhandler(int test)
{
try {
      // возбуждение исключительной ситуации типа int
      if(test==0) throw test;
      // возбуждение исключительной ситуации типа char
      if(test==1) throw 'a';
      // возбуждение исключительной ситуации типа double
      if(test==2) throw 123.23;
   }
   catch(int i) { // перехват исключительной ситуации типа int
      cout << "Перехвачен " << i << '\n';
   }
   catch(...) { // перехват исключительных ситуаций остальных типов
      cout << "Перехвачена ошибка!\n";
   }
}

int main()
{
   cout << "начало\n";
   Xhandler(0);
   Xhandler(1);
   Xhandler(2);
   cout << "конец";
   return 0;
}
```

На экран программа выводит следующее:

```
начало
Перехвачен 0
Перехвачена ошибка!
Перехвачена ошибка!
конец
```

Как показано в этом примере, использование инструкции **catch(…)** таким образом – хороший способ перехватывать те исключительные ситуации, которые вы не хотите обрабатывать явно. Кроме этого, путём перехвата всех исключительных ситуаций вы предотвращаете аварийное завершение программы из-за случайно необработанной исключительной ситуации.

3. В следующей программе показано, как ограничить число типов исключительных ситуаций, которые возбуждаются функцией:

```
/* Ограничение числа возбуждаемых функцией типов исключительных ситуаций
*/
#include <iostream>
using namespace std;

// Этой функцией могут возбуждаться только
// исключительные ситуации типов int, char и double
```

```
void Xhandler(int test) throw(int, char, double)
{
      // возбуждение исключительной ситуации типа int
      if(test==0) throw test;
      // возбуждение исключительной ситуации типа char
      if(test==1) throw 'a';
      // возбуждение исключительной ситуации типа double
      if(test==2) throw 123.23;
}

int main()
{
   cout << "начало\n";
   try {
      Xhandler(0); // попробуйте также передать в
                  // функцию Xhandler() значения 1 и 2
   }
   catch(int i) {
      cout << "Перехват int\n";
   }
   catch(char c) {
      cout << "Перехват char\n";
   }
   catch(double d) {
      cout << "Перехват double\n";
   }
   cout << "конец";
   return 0;
}
```

В этой программе функция **Xhandler()** может возбуждать только исключительные ситуации типа **int, char** и **double**. При попытке возбудить исключительную ситуацию другого типа произойдёт аварийное завершение программы (То есть будет вызвана функция **unexpected().**) Чтобы убедиться в этом, удалите из списка допустимых исключительных ситуаций тип **int** и повторите запуск программы.

Важно понимать, что ограничить типы возбуждаемых ситуаций можно только после того, как функция вызвана из блока **try**. То есть *внутри* функции блок **try** может возбудить любой тип исключительной ситуации, коль скоро она перехватывается *внутри* этой функции. Ограничения вступают в силу тогда, когда исключительная ситуация не перехвачена функцией.

4. Следующее небольшое изменение в функцию **Xhandler()** запрещает возбуждение любой исключительной ситуации:

```
//Эта функция НЕ может вызывать никаких исключительных ситуаций
void Xhandler(int test) throw()
{
/*Следующие инструкции больше не работают. Наоборот, попытка их выполнения ведёт к ненормальному завершению программы */

      //возбуждение исключительной ситуации типа int
      if  (test==0) throw test;
      //возбуждение исключительной ситуации типа char
      if  (test==1) throw ‘a’;
      //возбуждение исключительной ситуации типа double
      if  (test==2) throw 123.23; }
```

5. Вы уже знаете, что можно повторно возбудить исключительную ситуацию. Смысл этого в том, чтобы предоставить возможность обработки исключительной ситуации нескольким процедурам. Например, предположим, что одна процедура обрабатывает один аспект исключительной ситуации, а вторая – другой. Повторно исключительная ситуация может быть возбуждена только внутри блока **catch** (или любой функцией, которая вызывается из этого блока). Когда вы повторно возбуждаете исключительную ситуацию, она перехватывается не той же инструкцией **catch**, а переходит к другой, внешней к данной инструкции. В следующей программе иллюстрируется повторное возбуждение исключительной ситуации: возбуждается исключительная ситуация типа **char*.**

```
/* Пример повторного возбуждения исключительной ситуации одного и того же типа
*/
#include <iostream>
using namespace std;

void Xhandler()
{
   try {
      // возбуждение исключительной ситуации типа char *
      throw "привет";
   }
   // перехват исключительной ситуации типа char *
   catch(char *) {
      cout << "Перехват char * внутри функции Xhandler()\n";
      // повторное возбуждение исключительной ситуации
      // типа char *, но теперь уже не в функции Xhandler()
      throw;
   }
}

int main()
{
   cout << "начало\n";
   try {
      Xhandler();
   }
   catch(char *) {
      cout << " Перехват char * внутри функции main()\n";
   }
   cout << "конец";
   return 0;
}
```

На экран программа выводит следующее:

```
начало
Перехват char * внутри функции XhandlerO
Перехват char * внутри функции main()
конец
```

11.5. Обработка исключительных ситуаций, возбуждаемых оператором new

В главе 4 вы получили представление о том, что в соответствии с современной спецификацией оператора **new**, он возбуждает исключительную ситуацию при неудачной попытке выделения памяти. Поскольку в главе 4 об исключительных ситуациях мы еще не

знали, описание того, как они обрабатываются было отложено. Теперь настало время подробно исследовать ситуацию неудачной попытки выделения памяти с помощью оператора **new**.

В материале этого раздела атрибуты оператора **new** описаны так, как это определено в современном едином международном стандарте Standard C++. Как уже упоминалось в главе 4, с момента появления языка C++ точное определение действий, которые должны выполняться при неудачной попытке выделения памяти с помощью оператора **new,** менялось несколько раз. Сразу после разработки языка при неудачной попытке выделения памяти оператор **new** возвращал нуль, несколькими годами позднее – возбуждал исключительную ситуацию. Кроме того, неоднократно менялось имя этой исключительной ситуации. В конце концов было решено, что неудачная попытка выделения памяти с помощью оператора **new** по умолчанию будет возбуждать исключительную ситуацию, но по желанию в качестве опции можно возвращать нулевой указатель. Таким образом, оператор **new** реализовывался по-разному в разное время разными производителями компиляторов. Хотя в будущем все компиляторы должны быть выполнены в точном соответствии с требованиями стандарта Standard C++, сегодня это не так. Если представленные здесь примеры программ у вас не работают, проверьте по документации вашего компилятора то, как именно в нем реализован оператор **new**.

В соответствии со стандартом Standard C++, когда требование на выделение памяти не может быть выполнено, оператор **new** возбуждает исключительную ситуацию **bad_alloc**. При невозможности перехватить эту исключительную ситуацию программа завершается. Хотя для коротких программ такой алгоритм кажется вполне очевидным и понятным, в реальных приложениях вы должны не только перехватить, но и каким-то разумным образом обработать эту исключительную ситуацию. Для доступа к указанной исключительной ситуации в программу необходимо включить заголовок **<new>**.

Как уже упоминалось, в соответствии с требованиями стандарта Standard C++ при неудачной попытке выделения памяти допускается, чтобы оператор **new** возвращал нуль, а не возбуждал исключительную ситуацию. Такая форма оператора new может оказаться полезной при компиляции устаревших программ на современном компиляторе, а также при замене функций **malloc()** операторами **new**. Ниже представлена именно эта форма оператора **new**:

указатель **= new (nothrow)** ***тип***;

Здесь ***указатель*** - это указатель на переменную типа ***тип***. Форма оператора new с ключевым словом **nothrow** (без вызова исключительной ситуации) функционирует аналогично его прежней, изначальной версии. Поскольку в этом случае при неудачной попытке выделения памяти возвращается нуль, оператор **new** может быть легко "встроен" в старые программы и вам не нужно заботиться о формировании процедуры обработки какой бы то ни было исключительной ситуации. Тем не менее, в новых программах механизм исключительных ситуаций предоставляет вам гораздо более широкие возможности.

Примеры

1. В представленном ниже примере с оператором **new** использование блока **try/catch** дает возможность проконтролировать неудачную попытку выделения памяти.

```
#include <iostream>
#include <new>
```

```
using namespace std;
int main()
{
int *p;
try {
p = new int; // выделение памяти для целого
} catch (bad_alloc xa) {
cout < "Ошибка выделения памяти\n";
return 1;
}
for(*p = 0; *p < 10; (*p)++)
cout < *p < " ";
delete p; // освобождение памяти
return 0;
}
```

В данном примере, если при выделении памяти случается ошибка, она перехватывается инструкцией **catch.**

2. Поскольку в предыдущей программе при работе в нормальных условиях ошибка выделения памяти чрезвычайно маловероятна, в представленном ниже примере для демонстрации возможности возбуждения исключительной ситуации оператором **new** ошибка выделения памяти достигается принудительно. Процесс выделения памяти длится до тех пор, пока не произойдет ошибка.

```
#include <iostream>
#include <new>
using namespace std;
int mainf)
{
double *p;
// цикл будет продолжаться вплоть до исчерпания ресурса памяти
do {
try {
p = new double[100000];
} catch (bad_alloc xa) {
cout < "Ошибка выделения памяти\n";
return 1
}
cout < "Выделение памяти идет нормально\n";
} while (p);
return 0;
}
```

3. В следующей программе показано, как использовать альтернативную форму оператора **new** - оператор **new**(**nothrow**). Это переработанная версия предыдущей программы с принудительным возбуждением исключительной ситуации.

```
// Демонстрация работы оператора new(nothrow)
#include <iostream>
#include <new>
using namespace std;

int main()
{
   double *p;
   // цикл будет продолжаться вплоть до исчерпания ресурса памяти
   do {
      p = new(nothrow) double[100000];
      if(p) cout << "Выделение памяти идет нормально\n";
```

```
      else cout << "Ошибка выделения памяти\n";
   } while(p);
   return 0;
}
```

Как показано в этой программе, при использовании оператора **new** с ключевым словом **nothrow**, после каждого запроса на выделение памяти следует проверять возвращаемое оператором значение указателя.