Лабораторная работа №3
Шаблоны классов. Обработка исключительных ситуаций

Шаблоны классов

Шаблоны классов наряду с шаблонами функций поддерживают парадигму обобщенного программирования, то есть программирования с использованием типов в качестве параметров. Механизм шаблонов в C++ допускает применение абстрактного типа в качестве параметра при определении класса или функции. После того как шаблон класса определен, он может использоваться для определения конкретных классов. Процесс генерации компилятором определения конкретного класса по шаблону и его аргументам называется инстанцированием шаблона (template instantiation).

Рассмотрим, например, точку на плоскости. Для ее представления в задаче 3.2 мы разработали класс Point, в котором положение точки задавалось координатами x и y — полями типа double. Представим, что в другом приложении требуется задавать точки для целочисленной системы координат, то есть использовать поля типа int. Можно вообразить себе системы, в которых координаты точки имеют тип short или unsigned char. Так что же — определять для каждой из этих задач новый класс Point? Бьерна Страуструпа очень раздражала такая перспектива, и он добавил в C++ поддержку того, чем мы будем заниматься на этом семинаре.

Определение шаблона класса

Определение шаблонного (обобщенного, родового) класса имеет вид
`template <параметры_шаблона> class имя_класса { /* … */ };`

Например, определение шаблонного класса Point будет выглядеть так:

```
template <class T> class Point {
public:
    Point( T _x = 0, T _y = 0 ) : x( _x ), y( _y ) {}
    void Show() const { cout << "  (" << x << ", " << y << ")" <<
endl; };
private:
    T x, y;
};
```

Вы заметили, чем отличается это определение от определения обычного класса? Префикс template <class T> указывает, что объявлен шаблон класса, в котором T — некоторый абстрактный тип. То есть ключевое слово class в этом контексте задает вовсе не класс, а означает лишь то, что T — это параметр шаблона. Вместо T может использоваться любое имя. После объявления T используется внутри шаблона точно так же, как имена других типов. Язык позволяет вместо ключевого слова class перед параметром шаблона использовать другое — typename:

```
template < typename T> class Point { /* … */ };
```

В литературе встречаются оба стиля, но первый более распространен, так как ключевое слово `typename` появилось в языке C++ сравнительно недавно.

Определение встроенных методов внутри шаблона класса практически не отличается от записи в обычном классе. Но если определение метода выносится за пределы класса, синтаксис его заголовка усложняется. Покажем это на примере метода `Show`:

```
template <class T> class Point {     //Версия с внешним определением
                                       метода Show public:
    Point( T _x = 0, T _y = 0 ) : x( _x ), y( _y ) {}
    void Show() const;
 private:
    T x, y;
};
template <class T> void Point<T>::Show() const {
     cout << "  (" << x << ", " << y << ")" << endl;
 }
```

Обратите внимание на появление того же префикса `template <class T>`, который предварял объявление шаблона класса, а также на более сложную запись операции квалификации области видимости для имени `Show`: если раньше мы писали `Point::`, то теперь пишем `Point<T>::`, добавляя к имени класса список параметров шаблона в угловых скобках (в данном случае — один параметр `T`). На первых порах это сложно запомнить (что является причиной постоянных ошибок компиляции), но, написав пару десятков шаблонных классов, вы привыкнете…

Использование шаблона класса

При включении шаблона класса в программу никакие классы на самом деле не генерируются до тех пор, пока не будет создан экземпляр шаблонного класса, в котором вместо абстрактного типа `T` указывается некоторый конкретный тип. Такая подстановка приводит к *актуализации*, или *инстанцированию*, шаблона. Как и для обычного класса, экземпляр создается либо объявлением объекта, например:

```
Point<int> aPoint( 13, -5 );
```

либо объявлением указателя на актуализированный шаблонный тип с присваиванием ему адреса, возвращаемого операцией `new`, например:

```
Point<double>* pPoint = new Point<double>( 9.99, 3.33 );
```

Встретив такие объявления, компилятор генерирует код соответствующего класса.

Организация исходного кода

В многофайловом проекте определение шаблона класса обычно выносится в отдельный файл. В то же время для инстанцирования компилятором конкретного экземпляра шаблона класса необходимо, чтобы определение шаблона находилось *в одной единице трансляции* с данным экземпляром.

В связи с этим принято размещать *все определение* шаблонного класса в некотором заголовочном файле, например `Point.h`, и подключать его к нужным файлам с помощью директивы `#include`. Для предотвращения повторного включения этого файла, которое может возникнуть в многофайловом проекте, обязательно используйте «стражи включения».

Продемонстрируем эту технику на примере нашего класса `Point` (листинг 3.1).

Листинг 3.1. Шаблонный класс Point

```
//////////////////////////////////////////////// Point.h ///////
#ifndef POINT_H
#define POINT_H
using namespace std;
template <class T> class Point {
public:
    Point(T _x = 0, T _y = 0) : x(_x), y(_y) {}
    void Show() const;
 private:
    T x, y;
};
template <class T> void Point<T>::Show() const {
     cout << "  (" << x << ", " << y << ")" << endl;
}
#endif /* POINT_H */
//////////////////////////////////////////////// Main.cpp /////
#include <iostream>
#include "Point.h"
using namespace std;
int main() {
    Point<double> p1;                  // 1
    Point<double> p2(7.32, -2.6);      // 2
    p1.Show(); p2.Show();
    Point<int> p3(13, 15);             // 3
    Point<short> p4(17, 21);           // 4
    p3.Show(); p4.Show();
}
```

Обратите внимание на использование шаблонного класса `Point` клиентом `main`: в *строках 1* и *2* шаблон инстанцируется в конкретный класс `Point` с подстановкой вместо T типа double, в *строке 3* — в класс Point с подстановкой типа int, в *строке 4* — с подстановкой типа `short`.

Заметим, что наиболее широкое применение шаблоны классов нашли при создании контейнерных классов стандартной библиотеки шаблонов (STL), предназначенных для работы с такими стандартными структурами, как вектор, список, очередь, множество и т. д. Кстати, на семинаре 11 мы уже воспользовались одним из этих классов, а именно классом `vector`. Так что сейчас вы можете бросить беглый ретроспективный взгляд на пройденный материал, испытывая чувство глубокого удовлетворения от более тонкого понимания новой материи — шаблонных классов.

Параметры шаблонов

Параметрами шаблонов могут быть абстрактные типы или переменные встроенных типов, таких как `int`. Стандарт C++ допускает также использование параметровшаблонов, но это реализовано далеко не во всех компиляторах. *Абстрактные типы* в качестве параметров мы уже рассмотрели: при инстанцировании на их место подставляются аргументы либо встроенных, либо описанных программистом типов.

Переменные встроенных типов используются как параметры шаблонов, когда для шаблона предусматривается его настройка некоторой константой. Например, можно создать шаблон для массивов, содержащих `n` элементов типа `T`:

```
template <class T, int n> class Array { /*...*/ };
Array<Point, 20> ap;          // Создание массива из 20 элементов типа
Point
```

Приведем менее тривиальный пример использования параметров второго вида:

```
void f1() { cout << "I am f1()." << endl; }         // Директивы
препроцессора для
void f2() { cout << "I am f2()." << endl; }         // краткости
опущены
template<void (*pf)()> struct A { void Show() { pf(); } };
int main() {
     A<&f1> aa;
     aa.Show();    // вывод: I am f1().
     A<&f2> ab;
     ab.Show();    // вывод: I am f2().
}
```

Здесь параметр шаблона имеет тип указателя на функцию. При инстанцировании класса в качестве аргумента подставляется адрес соответствующей функции (адрес функции также является константой, создаваемой компилятором).

Естественно, у шаблона может быть несколько параметров. Например, ассоциативный контейнер `map` из библиотеки `STL` имеет следующее определение:

```
template <class Key, class T, class Compare = less<Key> > class map {
/*...*/ };
```

Из последнего примера видно, что параметры шаблона так же, как и параметры обычных функций, могут иметь значения по умолчанию. Обратите внимание на наличие пробела между двумя последними символами `>`. Если не поставить пробел, компилятор воспримет последовательность `>>` как операцию сдвига вправо и выдаст сообщение об ошибке, текст которого слабо проясняет ее истинную причину.

Специализация

Иногда возникает необходимость определить специализированную версию шаблона для некоторого конкретного типа одного из его параметров. Например, невозможно создать обобщенный алгоритм, проверяющий

отношение < (меньше) для двух аргументов типа T, который одновременно подходил бы и для числовых типов, и для традиционных Строк, завершающихся нулевым байтом. В таких случаях применяется специализация шаблона, имеющая следующую общую форму: `template <> class имя_класса <имя_специализированного_типа> { /* … */ };`

Например:

```
template <class T> class Sample {          // Общий шаблон
    bool Less(T) const; /*...*/ };
template <> class Sample<char*> {         // Специализация для char*
bool Less(char*) const; /*...*/ };
```

Если у класса-шаблона несколько параметров, возможна *частичная специализация*:

```
template <class T1, class T2> class Pair { /*...*/ };// Общий шаблон
// специализация, где для T2 установлен тип int:
template <class T1> class Pair <T1, int> { /*...*/ };
```

В любом случае общий шаблон должен быть объявлен прежде любой специализации.

Иногда есть смысл специализировать не весь класс, а какой-либо из его методов:

```
// обобщенный метод:
template <class T> bool Sample<T>::Less(T ob) const { /*...*/ };
// специализированный метод:
void Sample<char*>::Less(char* ob) const { /*...*/ };
```

Использование функциональных объектов для настройки шаблонных классов

Напомним, что *функциональным объектом* называется объект, для которого определена операция вызова функции `operator()`. Соответственно, класс, экземпляром которого является этот объект, называется *классом функционального объекта* или просто *функциональным классом*. Приведем пример (листинг 3.2).

Листинг 3.2. Использование функционального класса

```
#include <iostream>
using namespace std;
struct LessThan { // функциональный класс с единственной операцией
operator()
    bool operator() ( const int x, const int y ) { return x < y; }
};
int main() {
    LessThan l // объявлен объект lt функционального класса LessThan
    int a = 5, b = 9;
    if ( lt( a, b ) ) cout << a << " less than " << b << endl; // 2
    if ( LessThan()( a, b ) ) cout << a << " less than " << b << endl;
                                                        // 3 }
```

Как вы помните, `struct` — это вид класса, в котором все элементы по умолчанию открыты. Объект `lt` используется в операторе `if` (*оператор 2*) для вызова функции operator(), передавая ей аргументы a и b.

В *операторе 3* демонстрируется способ использования функционального объекта без его предварительного объявления. Запись `LessThan()` означает создание анонимного экземпляра класса `LessThan`, то есть функционального объекта. Запись LessThan() (a, b) означает вызов функции operator() с передачей ей аргументов a и `b`. Результаты выполнения обоих операторов `if` одинаковы.

Возможно, вы удивитесь, какой смысл использовать здесь класс `LessThan`, вместо того чтобы просто написать `if (a < b)`, и будете абсолютно правы. Но ситуация резко меняется, когда один из параметров шаблонного класса используется для настройки класса на некоторую стратегию.

Рассмотрим пример. Вас вызвал шеф и дал задание создать шаблонный класс `PairSelect`, предназначенный для выбора одного значения из пары значений, хранящихся в этом классе. Выбор выполняется в соответствии с критерием (стратегией), который передается как параметр шаблона. Параметр должен называться `class` Compare. Для начала нужно реализовать две стратегии: `LessThan` (первое значение меньше второго) и `GreaterThan` (первое значение больше второго).

Анализируя постановку задачи, вы приходите к заключению, что реализация стратегий в виде глобальных функций `LessThan()` и `GreaterThan()` быстро заведет в тупик, так как компилятор не позволит передать адреса функций на место параметра `class` Compare. Остается единственное решение — использовать функциональные классы (листинг 3.3).

Листинг **3.3**. Шаблонный класс PairSelect

```
#include <iostream>
using namespace std;
template <class T> struct LessThan {
    bool operator() ( const T& x, const T& y ) { return x < y; }
};
template <class T> struct GreaterThan {
    bool operator() ( const T& x, const T& y ) { return x > y; }
};
template <class T, class Compare>
class PairSelect {
public:
    PairSelect( const T& x, const T& y ) : a( x ), b( y ) {}
    void OutSelect() const {
        cout << ( Compare()( a, b ) ? a : b ) << endl;
    }
private:
    T a, b;
};
int main() {
    PairSelect<int, LessThan<int> > ps1( 13, 9 );
    ps1.OutSelect();                                    // вывод: 9
    PairSelect<double, GreaterThan<double> > ps2( 13.8, 9.2 );
    ps2.OutSelect();                                   // вывод: 13.8
```

}

Обратите внимание на использование функционального объекта Compare()(a, b) в методе OutSelect, а также на настройку шаблонного класса PairSelect при его инстанцировании. При первом инстанцировании (объявление объекта ps1) второму аргументу класса PairSelect передается функциональный класс LessThan<int>. При втором инстанцировании (объявление ps2) передается GreaterThan<double>.

Разработка шаблонного класса для представления разреженных массивов

Для закрепления материала разработаем шаблонный класс, предназначенный для представления *разреженных массивов* — массивов, в которых не все элементы фактически используются, присутствуют на своих местах или нужны.

Эта структура данных появилась в процессе решения научных и инженерных задач, в которых нужны многомерные массивы очень большого объема, но в то же время на каждом этапе обработки информации фактически используется только незначительная часть элементов массива. Например, это приложения, связанные с анализом матриц, или *электронные таблицы*, использующие матрицу для хранения формул, значений и строк, ассоциируемых с каждой из ячеек. Благодаря использованию разреженных массивов память для хранения каждого из элементов выделяется из пула свободной памяти только по мере надобности.

Для простоты изложения будем рассматривать одномерные разреженные массивы, так как все обсуждаемые идеи несложно применить и для реализации многомерных массивов. В среде приличных людей, так или иначе связанных с разреженными массивами, в ходу два термина: «*логический массив*» и «*физический массив*». Логический массив является воображаемым, а физический массив — это массив, фактически существующий в системе. Например, если вы определите разреженный массив:

SparseArr sa(1000000);

логический массив будет состоять из 1 000 000 элементов даже в случае, если этот массив в системе физически не существует. Если же фактически используется 100 элементов массива, то только они и будут занимать физическую память компьютера.

Обычно каждый элемент физического разреженного массива содержит как минимум два поля: логический индекс элемента и его значение. Для хранения физического массива, как правило, используют одну из динамических структур данных.

Задача 3.1. Шаблонный класс для разреженных массивов

Разработать шаблонный класс для представления разреженных одномерных массивов. Размер логического массива передавать через аргумент конструктора.

Класс должен обеспечивать хранение данных любого типа T, для которого предусмотрены конструктор по умолчанию, конструктор копирования и операция присваивания. Класс должен содержать операцию индексирования, возвращающую ссылку на найденный элемент. Если элемент с заданным индексом не найден, операция должна создать новый элемент с этим индексом и поместить его в массив. При необходимости добавить в класс другие методы. В клиенте main продемонстрировать использование этого класса.

Заметим, что согласно условию задачи нужно разработать очень примитивный класс с минимальной функциональностью. В реальных приложениях такой класс будет, конечно, содержать и другие методы, например, удаление из физического массива элемента с заданным индексом.

Выше было сказано, что каждый элемент физического массива должен содержать два поля: логический индекс элемента и его значение. Поэтому начнем с разработки серверного класса для представления одного элемента физического массива. Этот класс также должен быть шаблонным, иначе как же он же будет использоваться клиентом — шаблонным же классом разреженного массива? Результатом проработки этого вопроса может быть, например, следующий класс:

```
template <class DataT> class SA_item {
public:
    SA_item( long i, DataT d ) : index( i ), info( d ) {}
    long  index;
    DataT info;
};
```

Второй вопрос, который нужно решить до начала кодирования, — какую структуру данных мы выберем для хранения физического массива и какими средствами ее реализуем? Чаще всего используются линейные списки, бинарные деревья или структуры данных с хешированием индексов.

Линейный список имеет наихудшие показатели по времени поиска информации с заданным ключом (индексом), что может иметь существенное значение, если количество элементов в физическом массиве достаточно велико, но в то же время он наиболее прост для программирования. Более того, чтобы не отвлекаться на детали реализации и написать компактный код, мы воспользуемся контейнерным классом list из STL. Вообще-то контейнерным классам STL будет посвящен семинар 15, но у нас уже есть опыт использования класса `vector` на семинаре 11; так же и здесь, приведя минимально необходимые сведения о классе `list`, мы сможем воспользоваться многими его преимуществами.

Контейнерный класс `list` является шаблонным классом и реализован в STL в виде двусвязного списка, каждый узел которого содержит ссылки на последующий и предыдущий элементы. Для использования класса необходимо подключить заголовочный файл `<list>`. В классе есть конструктор по

умолчанию, создающий список нулевой длины. Можно добавить в конец имеющегося списка новый элемент с помощью метода `push_back`. Доступ к любому элементу списка осуществляется через *итератор* — переменную типа `list<T>::iterator`. Поскольку с понятием итератора вы еще не знакомы, скажем о нем несколько слов.

Проще всего рассматривать итератор как указатель на элемент списка. Он используется для просмотра списка в прямом или обратном направлении. В первом случае к итератору применяется операция инкремента, во втором — декремента.

В классе `list` есть два метода, позволяющие организовать просмотр всех элементов списка: `begin` возвращает указатель на первый элемент, `end` возвращает указатель на элемент, следующий *за* последним. Текущее значение итератора в цикле сравнивается со значением, полученным от метода `end`, с помощью операции `!=`, так как из-за произвольного размещения в памяти соседних элементов списка операция `<` для адресов элементов теряет смысл. Поясним применение класса `list` на примере:

```
#include <iostream>
#include <list>
using namespace std;
int main() {
     list<char> v1;
     v1.push_back( 'A' );
     v1.push_back( 'B' );
     v1.push_back( 'C' );
     list<char>::iterator i = v1.begin();
     list<char>::iterator n = v1.end();
     for ( ; i != n; ++i )
        cout << *i << ' '; // содержимое ячейки памяти, на которую
                           указывает i
    cout << endl;
}
```

Мы можем позволить себе не комментировать этот пример, так как уровень ваших знаний на текущий момент, безусловно, достаточен, чтобы понять все с полувзгляда. Теперь все готово, чтобы привести возможное решение задачи (листинг 3.4).

Листинг 3.4. Шаблонный класс для разреженных массивов

```
////////////////////////////////////////// SparseArr.h /////////
#ifndef SPARSE_ARR_H
#define SPARSE_ARR_H
#include <list>
using namespace std;
template <class DataT> class SA_item {
public:
    SA_item() : index( -1 ), info( DataT() ) {}
    SA_item( long i, DataT d ) : index( i ), info( d ) {}
    long index;
    DataT info;
```

```
};
template <class T> class SparseArr {
public:
     SparseArr( long len ) : length( len ) {}
     T& operator []( long ind );
     void Show( const char* );
 private:
     std::list<SA_item<T> > arr;                          // 1
     long length;
};
template <class T>
void SparseArr<T>::Show( const char* title ) {
     cout << "===== " << title << " =====\n";
     typename  list<SA_item<T> >::iterator i = arr.begin();
     typename  list<SA_item<T> >::iterator n = arr.end();
     for ( ; i != n; ++i )
        cout << i->index << "\t" << i->info << endl;
}
template <class T>
T& SparseArr<T>::operator[]( long ind ) {
    if ( ( ind < 0 ) || ( ind > length-1 ) ) {
         cerr << "Error of index: " << ind << endl;
         static SA_item<T> temp;
         return temp.info;
    }
    typename  list<SA_item<T> >::iterator i = arr.begin();
    typename  list<SA_item<T> >::iterator n = arr.end();
    for ( ; i != n; ++i )
        if ( ind == i->index ) return i->info;   // элемент найден
           // Элемент не найден, создаем новый элемент
arr.push_back( SA_item<T>( ind, T() ) );             // 2
    i = arr.end();
    return (--i)->info;
}
#endif   /* SPARSE_ARR_H */
////////////////////////////////////////////// Main.cpp ///////////
#include <iostream>
#include <string>
#include "SparseArr.h"
using namespace std;
int main() {
SparseArr<double> sa1( 2000000 );                    // 3
    sa1[127649] = 1.1;                               // 4
    sa1[38225] = 1.2;                                // 5
    sa1[2004056] = 1.3;                              // 6
    sa1[1999999] = 1.4;                              // 7
    sa1.Show("sa1");                                 // 8
    cout << "sa1[38225] = " << sa1[38225] << endl;   // 9
    sa1[38225] = sa1[93];                            // 10
    cout << "After the modification of sa1:\n";
    sa1.Show( "sa1" );
    SparseArr<string> sa2( 1000 );                   // 11
```

```
    sa2[333] = "Nick";
    sa2[222] = "Peter";
    sa2[444] = "Ann";
    sa2.Show( "sa2" );
    sa2[222] = sa2[555];
    sa2.Show( "sa2" );
}
```

Обратите внимание на следующие моменты. Поле `arr` (*оператор 1*) в шаблонном классе `SparseArr`, объявленное как объект шаблонного класса `list`, актуализированного шаблонным же параметром `SA_item<T>`, предназначено для хранения физического массива. Аргумент `T` здесь совпадает с параметром шаблонного класса `SparseArr`.

В операции индексирования `operator[]()` прежде всего проверяется, не выходит ли значение индекса `ind` за допустимые границы. Если выходит, формируется сообщение об ошибке, выводимое в поток `cerr`[1], после чего нужно либо прервать выполнение программы, либо вернуть некоторое приемлемое значение. В данном случае выбран второй вариант: создается временный объект `temp` с вызовом конструктора по умолчанию `SA_item()` и возвращается ссылка на его поле `info`.

Теперь обратим наши взоры на конструктор по умолчанию в классе `SA_item` — в первоначальном варианте класса, рассмотренном выше, этого конструктора не было. Здесь же он добавлен именно для использования в нештатных ситуациях: конструктор создает клон объекта физического массива со значением индекса, равным –1. Этим гарантируется, что созданный элемент нигде и никогда не будет использован. Таким образом, после вывода сообщения об ошибке программа продолжит свое выполнение.

Вряд ли стоит считать такое решение идеальным: ведь память будет замусориваться неиспользуемыми объектами. С другой стороны, терминальное прерывание работы программы вызовом `exit` тоже не назовешь эстетичным решением. Короче говоря, перед нами — классическая ситуация, требующая генерации и последующей обработки исключения, но эту тему мы рассмотрим чуть ниже. А пока вернемся к анализу работы операции индексирования.

Если с индексом все в порядке, то далее в цикле `for` ищется элемент физического массива с заданным индексом. В случае успеха возвращается ссылка на поле `info` найденного элемента. В случае неуспеха создается новый элемент физического массива и добавляется к списку `arr` (*оператор 2*). *При этом полю* `info` *присваивается значение, сформированное конструктором по умолчанию* `T()`. Затем вызывается метод `arr.end`, возвращающий указатель (итератор) на элемент, следующий *за последним* элементом списка. Чтобы

---

[1] cerr — объект класса ostream, представляющий *стандартное устройство для вывода со общений об ошибках*, который аналогично объекту cout направляет выводимые данные на терминал пользователя. Различие состоит в том, что вывод объекта cerr не буферизируется и отображается немедленно.

получить доступ к последнему элементу, применяется операция *префиксного* декремента `--i`. Полученное значение итератора позволяет вернуть из функции ссылку на поле `info` нового элемента.

В класс добавлен метод `Show`, который просто выводит в поток `cout` перечень элементов (индексы и значения) физического массива.

В клиенте `main` показано два варианта инстанцирования класса `SparseArr`. В первом случае (*оператор 3*) создается логический массив `sa1` из 2 000 000 элементов типа double. Поначалу он не содержит физических элементов. Выполнение *оператора 4* начнется с вызова операции sa1.operator[](127649). Так как элемента с указанным индексом в массиве нет, будет создан элемент, имеющий индекс 127649 и значение 0.0 (значение для типа `double` по умолчанию), и добавлен в конец массива. Операция индексирования вернет ссылку на поле `info` со значением 0.0, выполняемая следом операция присваивания изменит это значение на 1.1. Выполнение *операторов 5* и *7* аналогично, а вот при выполнении *оператора 6* будет обнаружена ошибка индексирования.

В *операторе 9* проверяется обращение к существующему элементу массива для его вывода в поток `cout`. Интересным является *оператор 10* — в результате его выполнения элемент `sa1[38225]` получит значение 0. Если вы внимательно читали предыдущие пояснения, такой результат не вызовет у вас вопросов. В итоге вывод на экран первой части программы (работа с массивом `sa1`) будет следующим:

```
Error of index: 2004056
===== sa1 =====
127649   1.1
38225    1.2
1999999  1.4
sa1[38225] = 1.2
After the modification of sa1:
===== sa1 =====
127649   1.1
38225    0
1999999  1.4
93       0
```

Во втором случае (*оператор 11*) создается логический массив `sa2` из 1000 элементов типа `string`. Действия по проверке его использования аналогичны предыдущим. Вывод на экран второй части программы будет следующим:

```
===== sa2 =====
333  Nick
222  Peter
444  Ann
===== sa2 =====
333  Nick
222
```

```
444  Ann
555
```

Обработка исключительных ситуаций

При решении предыдущей задачи мы столкнулись с проблемой — что делать методу класса (или, например, операции индексирования), если он обнаруживает некоторую ошибку, вызванную некорректным обращением клиента к методу (недопустимое значение индекса)? Инструментарий, которым мы пользовались до сих пор, начиная с семинара 7, позволял предпринять одно из следующих решений:

- прервать выполнение программы;
- возвратить значение, означающее «ошибка»;
- вывести сообщение об ошибке в поток `cerr` и вернуть вызывающей программе некоторое приемлемое значение, которое позволит ей продолжить работу.

Первый вариант решения не понравится пользователю программы, так как в самый неподходящий момент она будет «ломаться» без всякого уведомления о причине своей гибели. Второй вариант возможен лишь тогда, когда возвращаемое значение функции предназначено именно для кодирования статуса ее завершения. Так бывает далеко не всегда: например, в операции индексирования `operator[]()` класса SparseArr возвращаемое значение есть ссылка на поле `info` объекта. Ну и наконец, третий вариант решения тоже часто связан с труднорешаемыми вопросами: что есть «приемлемое значение» и почему программа продолжает свою работу независимо от обнаруженной ошибки?

С подобными проблемами часто сталкиваются разработчики промышленных библиотек классов. Автор библиотечного класса может обнаружить ошибки времени выполнения, но в общем случае не знает, как должен на них реагировать клиент. Для решения подобных проблем в C++ были введены средства генерации и обработки *исключений (exception)*. Заметим, что такими средствами пользуются не только при разработке библиотек. Например, в процессе выполнения конструктора класса может возникнуть какая-то нештатная ситуация (скажем, нехватка памяти). Поскольку конструктор не имеет возвращаемого значения, единственным способом для него уведомить об этом клиента также является генерация исключения.

Определение исключений

Для того чтобы работать с исключениями, необходимо:

- выделить *контролируемый блок* — составной оператор, перед которым записано ключевое слово try:

```
try {
    // фрагмент кода
```

```
}
```

- предусмотреть генерацию одного или нескольких исключений операторами `throw` внутри блока `try` или внутри функций, вызываемых из этого блока;

- разместить сразу за блоком `try` один или несколько обработчиков `catch`. Оператор throw, предназначенный для *генерации исключения*, имеет вид throw *выражение*,. Тип выражения определяет тип порождаемого исключения. При генерации исключения выполнение текущего блока прекращается, происходит поиск соответствующего обработчика и передача ему управления.

Синтаксис *обработчиков исключений* напоминает определение функции с одним параметром и именем catch:

```
catch (/* ... */) {
    // действия по обработке исключения
}
Объявление параметра обработчика возможно в одной из трех форм:
catch (Type) {           // Форма 1: обработка исключения типа Type
}
catch (Type info) {      // Форма 2: обработка исключения типа Type
                         //   с использованием значения info
}
catch (...) {            // Форма 3:обработка исключений всех типов
}
```

После обработки исключения управление передается первому оператору, находящемуся непосредственно за обработчиками. Туда же, минуя все обработчики, передается управление, если исключение в `try`-блоке не было сгенерировано.

Если обработчик не в состоянии полностью обработать ошибку, он может сгенерировать исключение повторно с помощью оператора `throw` без параметров. В этом случае предполагается наличие внешних объемлющих блоков, в которых может находиться другой обработчик для этого типа исключения.

Перехват исключений

Когда с помощью `throw` генерируется исключение, функции исполнительной библиотеки C++ выполняют следующие действия:

- создают копию параметра `throw` в виде статического объекта, который сохраняется до тех пор, пока исключение не будет обработано;

- в поисках подходящего обработчика *раскручивают стек* (об этом чуть ниже);

- передают объект и управление обработчику, имеющему параметр, совместимый по типу с этим объектом.

А что такое подходящий разработчик? Рассмотрим такой пример:

```
try {     throw E();
} catch ( H ) {
    // когда мы сюда попадем?
}
```

Обработчик будет вызван, если

- `H` и `E` — одного типа;
- `H` является открытым базовым классом для `E`;
- `H` и `E` — указатели либо ссылки и для них справедливо одно из предыдущих утверждений.

Следующая маленькая программка демонстрирует перехват исключений:

```
class A {
public:
    A() { cout << "Constructor of A\n"; }
    ~A() { cout << "Destructor of A\n"; }
};
class Error {};
class ErrorOfA : public Error {};
void foo() {
    A a;
    throw 1;
    cout << "This message is never printed" << endl;
}
int main() {
     try {
          foo();
        throw ErrorOfA();
    }
    catch( int )      { cerr << "Catch of int\n"; }
    catch( ErrorOfA ) { cerr << "Catch of ErrorOfA\n"; }
    catch( Error )    { cerr << "Catch of Error\n"; }
}
Она выдаст следующий результат:
Constructor of A
Destructor of A
Catch of int
```

Первым генерируется исключение `throw 1` внутри функции `foo`. Исполнительная система создает копию объекта «целая константа 1» и начинает поиск подходящего обработчика. Поскольку внутри `foo` ничего подходящего нет, поиск переносится вовне — в клиент `main`. Но (важная деталь!) перед тем, как покинуть тело функции `foo`, система вызывает деструкторы всех ее локальных объектов! Это и есть *раскрутка стека*.

В функции `main` нужный обработчик обнаруживается и дает о себе знать сообщением «Catch of int». Таким образом, генерация второго исключения `throw ErrorOfA()` здесь невозможна.

Поменяйте местами строки в блоке `try` и запустите программу снова. Теперь исключение throw `ErrorOfA()` будет сгенерировано и вы получите результат:

```
Catch of ErrorOfA
```

Интересно, что класс `ErrorOfA` совершенно пустой. Но этого достаточно, чтобы использовать его экземпляр (в данном случае анонимный) для генерации исключения. А теперь закомментируйте строку с обработчиком catch(ErrorOfA). Программа выдаст:

```
 Catch of Error
```

Неперехваченные исключения

Если исключение сгенерировано, но не перехвачено, вызывается стандартная функция `std::terminate`. Она будет вызвана и в том случае, если механизм обработки исключения обнаружит, что стек разрушен, или если деструктор, вызванный во время раскрутки стека, пытается завершить свою работу при помощи исключения.

По умолчанию terminate вызывает функцию abort.

Чтобы увидеть, что происходит при вызове этих функций, скомпилируйте и выполните следующую программу:

```
int main() {
    throw 5;    // произвольное значение
    return 0;
}
```

На нашем компьютере (Windows XP и Visual Studio 2005) на экран вываливается окно с сообщением об аварийном завершении программы:

```
    Debug Error!.. This application has requested the Runtime to
terminate it in an unusual way.
```

Вы можете заменить, если захотите, вызов функции `abort` вызовом своего обработчика. Это делается с помощью функции set_terminate. Например:

```
void SoftAbort() {
    cerr << "Program is terminated." << endl;
    exit( 1 );
}
int main() {
    set_terminate( SoftAbort );
    throw 5;
    return 0;
}
```

Проверьте, что завершение программы теперь действительно окажется более мягким.

Классы исключений. Иерархии исключений

Отметим, что, хотя язык позволяет генерировать исключения любого встроенного типа (как, например, типа `int` в рассмотренных выше примерах), в реальных программных системах такие исключения используются редко. Гораздо удобнее создавать специальные классы исключений (такие, как

ErrorOfA) и использовать в операторе throw либо объекты этих классов, либо анонимные экземпляры (через вызов конструктора класса).

При необходимости в этих классах можно передавать через параметры конструктора и сохранять для последующей обработки любую информацию о состоянии программы в момент генерации исключения. Другое преимущество этого подхода — возможность создания иерархии исключений. Например, исключения для математической библиотеки можно организовать следующим образом:

```
class MathError { /* ... */ };      // Базовый класс обработки ошибок
class Overflow : public MathError { /* ... */ }; // Класс ошибки
                                                    переполнения
class ZeroDivide : public MathError { /* ... */ }; // Класс ошибки
                                                    "деление на 0"
```

Если не полениться и добавить в базовый класс виртуальный метод ErrProcess, заместив его в производных классах версиями ErrProcess, ориентированными на обработку конкретного типа ошибки, это позволит задавать после блока try единственный обработчик catch, принимающий объект базового класса:

```
try {
 // генерация любых исключений MathError, Overflow, ZeroDivide, ...
}
catch (MathError& me) {
    me.ErrProcess()              // Обработка любого исключения
}
```

На этапе выполнения такой обработчик catch будет обрабатывать как исключения типа MathError, так и исключения любого производного типа, причем благодаря полиморфизму каждый раз будет вызываться версия метода ErrProcess, соответствующая именно данному типу исключения.

Организация исключений в виде иерархий исключительно важна, если ставится цель разработать легко модифицируемое программное обеспечение. Ведь при структуре кода, которую мы рассмотрели, добавление в систему нового вида исключения, входящего в существующую иерархию, вообще не потребует изменять написанные ранее фрагменты кода, предназначенные для перехвата и обработки исключений.

Спецификации исключений

В заголовке функции можно задать список типов исключений, которые она может прямо или косвенно порождать. Этот список приводится в конце заголовка и предваряется ключевым словом throw. Например: `void Func(int a) throw (Foo1, Foo2);`

Такое объявление означает, что Func может сгенерировать только исключения Foo1, Foo2 и исключения, являющиеся производными от этих типов, но не другие. Заголовок является интерфейсом функции, поэтому такое объявление дает пользователям функции определенные гарантии. Это очень

важно при использовании библиотечных функций, так как определения таких функций не всегда доступны.

Что произойдет, если функция вдруг нарушит взятые на себя обязательства и в ее недрах будет возбуждено исключение, не соответствующее списку спецификации исключений? Тогда система вызовет обработчик std::unexpected, который может попытаться сгенерировать свое исключение (это зависит от реализации), и если оно не будет противоречить спецификации, то продолжится поиск подходящего обработчика. В противном случае вызывается std::terminate.

Если вас не устраивает поведение `unexpected` по умолчанию, вы можете установить собственную функцию, которую он будет вызывать. Для этого нужно воспользоваться функцией set_unexpected.

Если спецификация исключений задана в виде `throw()`, это означает, что функция вообще не генерирует исключений. Отсутствие спецификации исключений в заголовке функции, то есть запись заголовка в привычном нам виде, означает, что функция может сгенерировать любое исключение.

Если заголовок функции содержит спецификацию исключений, то каждое объявление этой функции (включая определение) должно иметь спецификацию исключений с точно таким же набором типов исключений. Виртуальная функция может быть замещена в производном классе функцией с не менее ограничительной спецификацией исключений, чем ее собственная.

Спецификация исключений не является частью типа функции, поэтому `typedef` не может ее содержать. Например:

```
//  typedef int (*PF)() throw( A );    // ошибка!
```

Исключения в конструкторах

Исключения предоставляют единственную возможность передать информацию об обломе (срыве, крахе, фиаско), случившемся в процессе создания нового объекта. Рассмотрим пример кода (листинг 3.5), написанного, мягко говоря, некорректно, но именно это позволяет смоделировать ситуации, которые могут иметь место в реальной программе.

Листинг 3.5. Исключения в конструкторе

```
#include <iostream>
using namespace std;
class Vect {
 public:
    Vect(char);
    ~Vect() { delete [] p; }
    int& operator [] (int i) { return p[i]; }
    void Print();
 private:
```

```
     int* p;
     char size;
};
Vect::Vect(char n) : size(n) {
    p = new int[size];                        // 1
    for (int i = 0; i < size; ++i) p[i] = int();
}
void Vect::Print() {
    for (int i = 0; i < size; ++i) cout << p[i] << "  ";
    cout << endl;
} int main() {
      int x = -1;
      cout << "x = " << hex << x << endl;  // вывод в
                                            //  шестнадцатеричном
                                           // формате
    Vect a(3);
    a[0] = 0;  a[1] = 1;  a[2] = 2;  a.Print();
    Vect a1(200);                             // 2
    a1[10] = 5;  a1.Print();
}
```

В программе реализован класс `Vect`, предназначенный для создания и использования одномерных массивов типа `int` произвольного размера (размер передается через параметр конструктора класса).

Сначала несколько слов о возможной предыстории появления этого кода на свет. Руководитель проекта, для которого предназначен класс `Vect`, заявил, что размер массива никогда не превысит число 256. Проект (система) разрабатывается для «железа» с жуткими ограничениями на размер оперативной памяти. Программисты ведут борьбу за каждый байт! Человек, которому поручили реализацию класса Vect, решил хранить информацию о размере массива в поле `char size`. Ведь для типа char выделяется один байт, а диапазон возможных значений для восьмиразрядного двоичного числа составляет 0 ... 255.

В функции `main` созданный класс тестируется. Предварительно листинг 5 должен быть откомпилирован в отладочной конфигурации (Debug) проекта. Если запустить программу под отладчиком (клавиша F5), то программа выводит

x = ffffffff 0 1 2

после чего серьезно «ломается» — среда выполнения сообщает о недопустимом размере запрашиваемой памяти: 4 294 967 295 байт.

Выполнение по шагам показывает, что крах происходит при создании объекта `a1` (*оператор 2*). Если выполнить трассировку этого оператора, нажав клавишу F11, то можно достичь *оператора 1* в конструкторе `Vect` и увидеть, что значение $_{size}$, задающее размер запрашиваемой памяти, равно –56.

Откуда такое странное значение? Обычно для начинающих программистов непросто выяснить причину… Наш герой, видимо, тоже был

начинающим программистом. Более того, скорее всего, он невнимательно изучил Учебник [21], где на страницах

24–25 объясняется, что char — сокращенное обозначение типа signed char, в котором старший бит используется для представления знака числа. Поэтому диапазон представимых чисел для типа `char` составляет –128 … +127.

Как же будет интерпретироваться в этом случае десятичное число 200? Преобразовав его в двоичный эквивалент, получаем 11001000. Вы видите, что старший разряд равен единице?.. Это означает, что компьютер воспримет его как –1001000.

Обратный перевод в десятичную систему будет несколько сложнее. Напомним, что отрицательные целые числа хранятся в памяти компьютера в *дополнительном коде*, который получается инверсией всех разрядов с последующим прибавлением единицы. Значит, для обратного перевода нужно сначала вычесть единицу, а потом инвертировать все разряды. В итоге получим число –56.

Если теперь выполнить трассировку *оператора 1*, нажав клавишу F11, то отладчик остановится перед заголовком библиотечной реализации операции `new: void *__CRTDECL operator new[](size_t count)`

а значение аргумента `count` будет равно уже не –56, а 4 294 967 295. Если перевести это десятичное число в шестнадцатеричный эквивалент, получим ffff ffff. Для типа size_t, эквивалентного типу unsigned int, это максимально возможное значение 32-разрядного двоичного числа. Для типа `signed int` это значение представляет –1 (см. результат вывода программы для переменной `x`).

Далее операция `new` завершается аварийно, будучи не в состоянии выделить 4 294 967 295 байт.

Итак, мы нашли ошибку в реализации класса. Исправить ошибку несложно — нужно все char заменить на unsigned char (проверьте это). Но не будем спешить. Дело в том, что мы имеем прекрасную модель ненормативного поведения операции `new` и для дальнейших экспериментов она нам еще пригодится.

Приведенный пример показал, что, если вызываемые в теле конструктора класса функции или операции могут завершиться аварийно, необходимо информировать об этом клиента (функцию, в теле которой создается объект класса). Использование механизма исключений дает элегантное решение проблемы. Изменим код конструктора класса на следующий:

```
Vect::Vect(char n) : size( n ) {
     try { p = new int[size]; }
     catch(...) {
     throw "ErrorVectConstr"; }
     for ( int i = 0; i < size; ++i ) p[i] = int();
}
```

Внесите также изменения в функцию main, использующую объекты класса Vect:

```
int main() {
     try {
        int x = -1;
        cout << "x = " << hex << x << endl;
        Vect a(3);
        a[0] = 0;  a[1] = 1;  a[2] = 2;  a.Print();
        Vect a1(200);
        a1[10] = 5;  a1.Print();
    }
    catch( const char* except_name ) {  cout << "Error: " <<
except_name << endl; }
}
```

Для проверки работы новой версии программы необходимо откомпилировать листинг 3.5 в выпускной конфигурации (Release) проекта. Дело в том, что по непонятным причинам операция new в среде Visual Studio 2005 генерирует исключения только в выпускной конфигурации проекта.

Исключения в деструкторах

Если деструктор, вызванный во время раскрутки стека, попытается завершить свою работу при помощи исключения, система вызовет функцию terminate. На этапе отладки программы это допустимо, но в готовом продукте появление окон сообщений об ошибках должно быть сведено к минимуму.

Отсюда наиважнейшее требование к деструктору: *ни одно из исключений, которое могло бы появиться в процессе работы деструктора, не должно покинуть его пределы*! Чтобы выполнить это требование, придерживайтесь двух правил.

1. Никогда не генерируйте исключения в теле деструктора с помощью throw.

2. Если финальные действия в деструкторе связаны с вызовом других функций, относительно которых у вас нет гарантий отсутствия генерации исключений, инкапсулируйте эти действия в одном методе, например Destroy, и вызывайте его в блоке try:

```
T::Destroy() {
    // код, который может генерировать исключения
}
T::~T() {                          // деструктор
    try { Destroy(); }
    catch(…) { /* … */ }
}
```

Мы завершили рассмотрение вопросов, связанных с обработкой исключений. Рекомендуем вам доработать листинг 3.4, предусмотрев в реализации операции индексирования генерацию исключения для ошибочного значения индекса.

Теперь рассмотрим решение задачи, в которой требуется использовать и шаблонные классы, и обработку исключений.

Задача 3.2. Шаблонный класс векторов

Разработать шаблонный класс Vect для представления динамических одномерных массивов (векторов). Класс должен обеспечивать хранение данных любого типа T, для которого предусмотрены конструктор по умолчанию, конструктор копирования и операция присваивания. Класс должен содержать:

- *конструктор по умолчанию, создающий вектор нулевого размера;*
- *конструктор, создающий вектор заданного размера;*
- *операцию индексирования, возвращающую ссылку на соответствующий элемент вектора;*
- *метод, добавляющий элемент в произвольную позицию вектора;* ‰ *метод, добавляющий элемент в конец вектора;* ‰ *метод, удаляющий элемент из конца вектора.*

При необходимости добавить в класс другие методы. Предусмотреть генерацию и обработку исключений для возможных ошибочных ситуаций. В клиенте main продемонстрировать использование этого класса.

Одним из принципиальных вопросов при разработке контейнерного класса является вопрос реализации хранения элементов контейнера, или, другими словами, выбор подходящей структуры данных (массив, список, бинарное дерево и т. п.). В данном случае мы остановим наш выбор на динамическом массиве, размещаемом в памяти посредством операции `new`, поскольку это наиболее простое решение. После размещения адрес первого элемента вектора будет запоминаться в поле `T* first`, а адрес элемента, следующего за последним, — в поле `T* last`. С учетом того, что используемый в классе метод `size` возвращает количество элементов в векторе, размещение контейнера в памяти поясняется на рис. 3.1

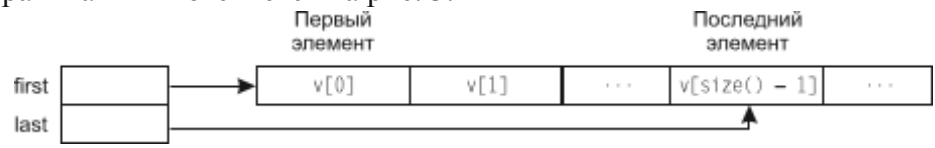

Рис. 3.1. Размещение вектора v в памяти

При решении этой задачи мы постараемся сделать класс `Vect` как можно более похожим (с точки зрения его интерфейса и, частично, реализации) на контейнерный класс `std::vector` из библиотеки STL (с учетом ограничений, вытекающих из постановки задачи). Поэтому в состав интерфейса будут включены методы:

– `void insert(`T* _P, const T& _x) — вставка элемента в позицию _P;

– `void push_back(const T& _x)` — вставка элемента в конец вектора;

– `void pop_back()` — удаление элемента из конца вектора;

– `T* begin()` — получение указателя на первый элемент;

– `T* end()` — получение указателя на элемент, следующий за последним;

– size_t size() — получение размера вектора (тип std::size_t является синонимом типа unsigned int).

Эти методы имитируют интерфейс класса `std::vector`. Уточним, что вместо типа iterator, имеющегося в std::vector, здесь используется указатель T*, но концептуально это одно и то же. Целью такого подражания является психологическая подготовка наших читателей к более легкому восприятию контейнерных классов STL (семинар 15).

К двум конструкторам, требующимся по заданию, добавим конструктор копирования, чтобы обеспечить возможность передачи объектов класса в качестве аргументов для любой внешней функции.

По некоторым параметрам наш класс `Vect`, однако, будет превосходить класс std::vector. Для целей отладки и обучения мы хотим визуализировать работу таких методов класса, как конструктор копирования и деструктор, путем вывода на терминал соответствующих сообщений. Чтобы эти сообщения были более информативны, мы снабдим каждый объект класса так называемым отладочным именем (поле markName), которое позволит распознавать источник сообщения.

Трудно переоценить, какую пользу приносят эти сообщения начинающим программистам, помогая прочувствовать скрытые механизмы функционирования класса. Для более опытных читателей эти средства могут оказаться полезными при поиске неочевидных ошибок в программе.

Задание требует также п*редусмотреть генерацию и обработку исключений для возможных ошибочных ситуаций. Наиболее очевидными являются: ошибка индексирования, когда клиент пытается получить доступ к несуществующему элементу вектора, и ошибка удаления несуществующего элемента из конца вектора — когда вектор пуст*. В принципе возможна еще одна нештатная ситуация, связанная с нехваткой памяти, когда операция $_{new}$ завершается аварийно. Но мы рассмотрели выше, как решать эту проблему, и

сейчас ради краткости изложения будем полагать, что обладаем неограниченной памятью.

Для обработки исключительных ситуаций создадим иерархию классов во главе с базовым классом VectError. Производный класс VectRangeErr будет предназначен для обработки ошибок индексирования, а производный класс `VectPopErr` — для обработки ошибки удаления несуществующего элемента. В листинге 3.6 приводится код предлагаемого решения задачи.

Листинг 3.6. Шаблонный класс векторов

```
//////////////////////////////////////////// VectError.h ////////
#ifndef _VECT_ERROR_
#define _VECT_ERROR_
#include <iostream>
#define DEBUG
class VectError {
public:
    VectError() {}
    virtual void ErrMsg() const { std::cerr << "Error with Vect
object.\n"; }
    void Continue() const {
#ifdef DEBUG
        std::cerr << "Debug: program is being continued.\n";
#else
        throw;
#endif
    }
};
class VectRangeErr : public VectError {
public:
    VectRangeErr(int _min, int _max, int _actual) :
                  min(_min), max(_max), actual(_actual) {}
    void ErrMsg() const {
        std::cerr << "Error of index: ";
        std::cerr << "possible range: " << min << " - " << max << ",
";
        std::cerr << "actual index: " << actual << std::endl;
        Continue();
    }
private:
    int min, max;
    int actual;
};
class VectPopErr : public VectError {
public:
    void ErrMsg() const { std::cerr << "Error of pop\n"; Continue();}
};
#endif /* _VECT_ERROR_ */
/////////////////////////////////////////// Vect.h //////////////
#ifndef _VECT_
#define _VECT_
#include <iostream>
```

```
#include <string>
#include "VectError.h"
using namespace std;
template<class T> class Vect {      // ------------------------
Template class Vect
public:
    explicit Vect() : first(0), last(0) {}
    explicit Vect(size_t _n, const T& _v = T()) {
        Allocate( _n );
        for (size_t i = 0; i < _n; ++i) *(first + i) = _v;
}
    Vect( const Vect& );                  // конструктор копирования
    Vect& operator =(const Vect&);        // операция присваивания
    ~Vect() {
#ifdef DEBUG
        cout << "Destructor of " << markName << endl;
#endif
        Destroy();
        first = 0, last = 0;
    }
    void mark(std::string& name) { markName = name; }// установить
отладочное имя
   std::string mark() const { return markName; }   // получить
отладочное имя
    size_t size() const;                   // получить размер вектора
    T* begin() const { return first; }     // получить указатель на 1-й
элемент
    T* end() const { return last; }        // получить указатель на
элемент,
                                           // следующий за последним
    T& operator[](size_t i);               // операция индексирования
    void insert(T* _P, const T& _x);       // вставка элемента в
позицию _P
    void push_back(const T& _x); // вставка элемента в конец вектора
   void pop_back();              // удаление элемента из конца вектора
  void show() const;            // вывод в cout содержимого вектора
protected:
    void Allocate( size_t _n ) {
         first = new T [_n * sizeof(T)];
         last = first + _n;
    }
    void Destroy() {
        for ( T* p = first; p != last; ++p )   p->~T();
        delete [] first;
    }
    T* first;        // указатель на 1-й элемент
    T* last;        // указатель на элемент, следующий за последним
    std::string markName;
};
template<class T>
Vect<T>::Vect(const Vect& other) {   // -------------------
Конструктор копирования
```

```
    size_t n = other.size();
    Allocate( n );
    for (size_t i = 0; i < n; ++i) *(first + i) = *(other.first + i);
    markName = string("Copy of ") + other.markName;
#ifdef DEBUG
    cout << "Copy constructor: " << markName << endl;
#endif
}
template<class T>
Vect<T>& Vect<T>::operator =(const Vect& other) { // --- Операция
присваивания
    if (this == &other) return *this;
    Destroy();
    size_t n = other.size();
    Allocate(n);
    for (size_t i = 0; i < n; ++i) *(first + i) = *(other.first + i);
    return *this;
}
template<class T>
size_t Vect<T>::size() const {  // ------Получение размера вектора
    if (first > last) throw VectError();     return (0 == first ? 0 :
last - first);
}
template<class T>
T& Vect<T>::operator[](size_t i) { //  Операция доступа по индексу
    if(i < 0 || i > ( size() - 1 ))
        throw VectRangeErr(0, last - first - 1, i);
    return ( *( first + i ) );
}
template<class T>   // Вставка элемента со значением _x в позицию _P
void Vect<T>::insert(T* _P, const T& _x) {
    size_t n = size() + 1;              // новый размер
    T* new_first = new T [n * sizeof(T)];
    T* new_last = new_first + n;     size_t offset = _P - first;
    for (size_t i = 0; i < offset; ++i) *(new_first + i) = *(first +
i);
    *( new_first + offset ) = _x;
    for (size_t i = offset; i < n; ++i) *(new_first + i + 1) = *(first
+ i);
    Destroy();
    first = new_first;
    last = new_last;
}
template<class T>
void Vect<T>::push_back(const T& _x) { // ---- Вставка элемента в
конец вектора
    if (!size()) { Allocate(1); *first = _x; }
    else insert( end(), _x );
}
template<class T>
void Vect<T>::pop_back() { // --- Удаление элемента из конца вектора
    if(last == first) throw VectPopErr();
```

```
    T* p = end() - 1;
    p->~T();
    last--;
}
template<class T>
void Vect<T>::show() const { // --- Вывод в cout содержимого вектора
    cout << "\n===== Contents of " << markName << "=====" << endl;
    size_t n = size();
    for (size_t i = 0; i < n; ++i) cout << *(first + i) << " ";
    cout << endl;
}
#endif /* _VECT_ */
////////////////////////////////////////////// Main.cpp ///////////
#include "Vect.h"
#include <iostream>
#include <string>
using namespace std;
template<class T> void SomeFunction( Vect<T> v ) {
    std::cout << "Reversive output for " << v.mark() << ":" << endl;
    size_t n = v.size();
    for (int i = n - 1; i >= 0; --i) std::cout << v[i] << " ";
    std::cout << endl;
}
int main() {
   std::string vv1 = "v1";
   std::string vv2 = "v2";
   std::string vv3 = "v3";
   std::string vv4 = "v4";
   try {
        string initStr[5] = { "first", "second", "third", "fourth",
"fifth" };
        Vect<int> v1(10);  v1.mark(vv1);
        size_t n = v1.size();
        for (int i = 0; i < n; ++i) v1[i] = i + 1;
        v1.show();
        SomeFunction(v1);
        try {
            Vect<string> v2(5);
            v2.mark(vv2);
            size_t n = v2.size();
            for (int i = 0; i < n; ++i) v2[i] = initStr[i];
            v2.show();
            v2.insert(v2.begin() + 3, "After_third");
            v2.show();              cout << v2[6] << endl;
            v2.push_back("Add_1"); v2.push_back("Add_2");
v2.push_back("Add_3");
            v2.show();
            v2.pop_back(); v2.pop_back();
            v2.show();
        }
        catch (VectError& vre)  { vre.ErrMsg(); }
        try {
```

```
            Vect<int> v3;
      v3.mark(vv3);
            v3.push_back(41); v3.push_back(42); v3.push_back(43);
            v3.show();
            Vect<int> v4;
            v4.mark(vv4);
            v4 = v3;
            v4.show();
            v3.pop_back();    v3.pop_back();
            v3.pop_back();    v3.pop_back();
            v3.show();
        }
        catch (VectError& vre)  { vre.ErrMsg(); }
    }
    catch (...) { cerr << "Epilogue: error of Main().\n"; }
}
```

Наши комментарии будут следовать в порядке размещения текста модулей. **Модуль** `VectError.h` содержит иерархию классов исключений. В базовом классе VectError определены два метода. Виртуальный метод ErrMsg обеспечивает вывод по умолчанию сообщения об ошибке (в производных классах этот метод замещается конкретными методами). Метод `Continue` определяет стратегию продолжения работы программы после обнаружения ошибки. Стратегия зависит от конфигурации программы. Информация о конфигурации кодируется наличием или отсутствием единственной директивы в начале файла: #define DEBUG.

Если лексема `DEBUG` определена, программа компилируется в *отладочной конфигурации*, если не определена — в *выпускной конфигурации*. Для метода `Continue` это означает, что

- в отладочной конфигурации выводится сообщение «Debug: program is being continued», после чего работа программы продолжается;
- в выпускной конфигурации повторно возбуждается (оператор `throw`) исключение, которое было первопричиной цепочки событий, завершившихся вызовом данного метода.

В производных классах VectRangeErr и VectPopErr метод ErrMsg переопределяется с учетом вывода информации о конкретной ошибке. После вывода вызывается метод Continue базового класса.

Модуль `Vect.h` содержит определение шаблонного класса `Vect`. Отметим наиболее интересные моменты.

Для управления выделением и освобождением ресурсов класс снабжен методами

Allocate и Destroy. Они размещены в защищенной части класса, так как относятся к его реализации. Напомним, что операция `new T [n]` (`n` _ количество элементов, которое мы хотим поместить в массив) не только

выделяет память, но и инициализирует элементы путем вызова `T()` — конструктора по умолчанию для типа `T`. Поэтому в методе `Destroy` сначала в цикле вызываются деструкторы `~T()` всех элементов массива, а потом операцией `delete` освобождается память.

Обратите внимание, что в заголовке цикла `for` условие его продолжения записано в виде p != last, а не p < last. Здесь мы следуем традиции STL. Подобный способ проверки обусловлен тем, что для произвольной структуры данных (например, для списка) соседние элементы не обязаны занимать подряд идущие ячейки памяти, а могут быть разбросаны в памяти как угодно.

Деструктор и конструктор копирования содержат вывод сообщений типа «Здесь был я!», дополненных печатью содержимого `markName`. Весь этот вывод осуществляется *только в отладочной конфигурации* программы. Операция присваивания (верная спутница конструктора копирования) реализована стандартным способом.

В операции доступа по индексу `operator[]()` служба внутренней охраны (оператор `if`) проверяет, не несет ли индекс с собой что-либо взрывоопасное (значение, лежащее вне пределов допустимого диапазона). Если индекс оказывается неблагонадежным, служба бьет тревогу, вызывая исключение типа `VectRangeErr`. Вызов происходит через анонимный экземпляр класса с передачей конструктору трех аргументов: минимальной и максимальной границы и текущего значения индекса.

В методе получения размера вектора `size` предусмотрена проверка на лояльность взаимоотношений между `first` и `last`. Вообще-то непонятно, с каких это дел `first` может оказаться больше, чем `last`? Да, в нормально функционирующем классе это невозможно. Но в случае каких-либо ошибок или сбоев указанная проверка весьма полезна. При несоблюдении указанного условия генерируется исключение типа

VectError.

Метод `insert` (вставка элемента со значением `_x` в произвольную позицию `_P`) работает следующим образом. Создается новый вектор размером на единицу больше существующего вектора; адреса размещения нового вектора сохраняются в локальных переменных `new_first, new_last`. Затем в новый вектор копируется первая часть существующего вектора, начиная с первого элемента и заканчивая элементом, предшествующим элементу с адресом `_P`.

Поскольку доступ к элементам осуществляется через смещение относительно указателя `first`, перед копированием требуется вычислить `offset` — смещение, соответствующее адресу _P[2]. Затем элементу с адресом new_first + offset присваивается значение `_x`. После этого оставшаяся часть существующего вектора копируется в новый вектор, начиная с позиции

[2] Напомним, что разность двух указателей — это разность их значений, деленная на размер типа в байтах. Например, в применении к массивам разность указателей на второй и седьмой элементы равна 5.

new_first + offset + 1. Вот и все. Осталось освободить ресурсы, занятые существующим вектором (`Destroy`), и назначить полям first и last новые значения.

Метод `push_back` (вставка элемента со значением `_x` в конец вектора) получился очень простым благодаря использованию серверной функции `insert`. Если вектор пуст, то выделяются ресурсы для хранения одного элемента и этому элементу присваивается значение `_x`. В противном случае вызывается `insert` для вставки элемента в позицию, определенную с помощью `end`.

В методе `pop_back` (удаление элемента из конца вектора) проверяется, не пуст ли вектор. В пустом векторе `first = last = 0`. Если это так, удалять нечего и, значит, клиент ошибся, затребовав такую операцию. Следовательно, надо ему об этом сообщить: генерируется исключение типа `VectPopErr`. В противном случае определяется указатель на последний элемент вектора, для него вызывается деструктор `~T()` и значение `last` уменьшается на единицу (учитывая арифметику указателей).

Модуль Main.cpp содержит определение глобальной шаблонной функции SomeFunction и определение функции `main`.

Функция `SomeFunction` обеспечивает реверсивный вывод содержимого вектора в поток `cout`. Попутно она позволяет проверить передачу объектов конкретного класса Vect в качестве аргументов функции.

Функция `main` предназначена для тестирования класса `Vect`. Советуем внимательно изучить ее многослойную архитектуру: внешний блок $_{try}$ содержит в себе два внутренних блока `try`. И это не случайно! Первый внутренний блок `try` предназначен для тестирования операций с объектом `v2`, среди которых есть операция, вызывающая ошибку индексирования. Второй блок `try` работает с объектом `v3`, причем одна из операций вызывает ошибку удаления несуществующего элемента.

Обратите внимание, что обработчики `catch` в обоих случаях имеют параметром объект базового класса `VectError`, но благодаря полиморфизму они будут перехватывать и исключения производных классов `VectRangeErr` и `VectPopErr`, так что в результате будет вызываться конкретный метод `ErrMsg` для данного типа ошибки! Вспомним, что после вывода сообщения об ошибке метод `ErrMsg` вызывает метод Continue, работа которого зависит от конфигурации программы.

В отладочной конфигурации `Continue` сообщает о продолжении работы программы и возвращает управление клиенту. В выпускной конфигурации `Continue` возбуждает исключение повторно. Именно для того, чтобы его поймать, и служит внешний блок try с обработчиком catch(…) в основной программе.

Откомпилируйте и выполните программу, чтобы увидеть, как она работает. Исходный текст настроен на работу в отладочной конфигурации. Всмотритесь внимательно в отладочные сообщения конструктора копирования и деструктора. А теперь закомментируйте директиву `#define DEBUG` и выполните программу после компиляции заново. Почувствуйте разницу!

Итоги

1. Шаблоны классов поддерживают парадигму обобщенного программирования — программирования с использованием типов в качестве параметров. Определение конкретного класса по шаблону класса и аргументам шаблона называется инстанцированием (актуализацией) шаблона.
2. Параметрами шаблонов могут быть абстрактные типы или переменные встроенных типов.
3. Для настройки шаблонного класса на некоторую стратегию возможно использование классов функциональных объектов.
4. Обработка исключительных ситуаций позволяет разделить проблему обработки ошибок на две фазы: генерацию исключения в случае нарушения каких-то заданных условий и последующую обработку сгенерированного исключения.
5. Фазы генерации и обработки исключений обычно разнесены в программе по разным компонентам (модулям, функциям). Это дает существенные преимущества, так как в месте возникновения ошибки (серверная функция) часто нет возможности корректно ее обработать, а клиентская функция такими возможностями обычно располагает.
6. Для конструктора класса использование исключений является единственным способом сообщить клиенту об ошибках или сбоях, случившихся в процессе конструирования объекта.
7. Для работы с исключениями необходимо: а) описать контролируемый блок; б) предусмотреть генерацию одного или нескольких исключений внутри блока или внутри функций, вызываемых из этого блока; в) разместить сразу за блоком один или несколько обработчиков исключений.
8. В качестве типов исключений, генерируемых оператором `throw`, целесообразно использовать определенные программистом классы исключений. Эти классы полезно выстраивать в иерархию, что позволяет создавать программы, для которых модификация в связи с обработкой новых типов ошибок оказывается менее трудоемкой.
9. Не генерируйте исключения в теле деструктора. Если это могут сделать другие вызываемые функции, примите все меры, чтобы исключение не покинуло деструктор.

Задания

Требуется создать шаблон некоторого целевого класса `A`. В каждом варианте уточняются требования к реализации — указанием на применение некоторого серверного класса `B`. Это означает, что объект класса `B`

используется как элемент класса `A`. В качестве серверного класса может быть указан либо класс, созданный программистом в рамках того же задания, либо класс стандартной библиотеки.

Варианты целевых и серверных классов, создаваемых программистом, приведены в табл. 3.1. Информацию о работе с динамическими структурами данных см. в Учебнике (с. 114–127) и в семинаре 09.

Таблица 3.1. Варианты целевых или серверных классов

Имя класса	**Назначение**
Vect	Одномерный динамический массив
List	Двунаправленный список
Stack	Стек
BinaryTree	Бинарное дерево
Queue	Односторонняя очередь
Deque	Двусторонняя очередь (допускает вставку и удаление из обоих концов)
Set	Множество (повторяющиеся элементы в множество не заносятся; элементы в множестве хранятся отсортированными)
SparseArray	Разреженный массив

Если вместо серверного класса указан динамический массив, это означает, что для хранения элементов в целевом классе используется массив, размещаемый с помощью операции `new`. Во всех вариантах необходимо предусмотреть генерацию и обработку исключений для возможных ошибочных ситуаций.

Во всех вариантах показать в клиенте `main` использование созданного класса, включая ситуации, приводящие к генерации исключений. Показать инстанцирование шаблона для типов `int`, `double`, `std::string`. Варианты заданий приведены в табл. 3.2.

Таблица 3.2. Варианты заданий

Вариант	Целевой шаблонный класс	Реализация с применением
1	Vect	std::list
2	List	—
3	Stack	Динамический массив
4	Stack	Vect
5	Stack	List
6	Stack	std::vector
7	Stack	std::list
8	BinaryTree	—
9	Queue	Vect
10	Queue	List
11	Queue	std::list
12	Deque	Vect
13	Deque	List
14	Deque	std::list
15	Set	Динамический массив
16	Set	Vect
17	Set	List
18	Set	std::vector
19	Set	std::list
20	SparseArray	