МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ РОССИЙСКОЙ ФЕДЕРАЦИИ
Озерский технологический институт — филиал НИЯУ МИФИ

Кафедра прикладной математики

Вл. Пономарев

ПРАКТИКУМ

по объектно-ориентированному программированию

Учебно-методическое пособие

Часть 3. Шаблоны и исключения

Озерск, 2019 г.

УДК 681.3.06
П 56

Вл. Пономарев. Практикум по ООП. Учебно-методическое пособие по объектно-ориентированному программированию. Часть 3. Шаблоны и исключения. Озерск: ОТИ НИЯУ МИФИ, 2019. — 35 с.

В пособии излагается, как выполнять практические работы по дисциплине «Объектно-ориентированное программирование».

В качестве основного материала при выполнении практических работ пособие предназначено для студентов, обучающихся по направлению подготовки 09.03.01 «Информатика и вычислительная техника» и специальности 09.05.01 «Применение и эксплуатация автоматизированных систем специального назначения».

Рецензенты:
1.
2.

УТВЕРЖДЕНО
Редакционно-издательским
Советом ОТИ НИЯУ МИФИ

Содержание

Общие цели занятий

В ходе практических работ изучаются основы использования классов в рамках методологии объектно-ориентированного программирования.

В этой части работ рассматриваются следующие темы:

1) пользовательские шаблоны функций;

2) пользовательские шаблоны классов;

3) стандартные шаблоны STL;

4) обработка исключений.

К практическим работам приписаны контрольные вопросы и упражнения. Контрольные вопросы могут быть заданы преподавателем в ходе защиты работы, однако преподаватель может задавать и другие вопросы, не указанные в списке.

Для защиты знаний и навыков, полученных в ходе выполнения практических работ студент должен иметь тетрадь (12-18 листов). Для каждой выполненной работы в тетрадь записывается отчет.

Отчет по работе начинается с заголовка:

1. Фамилия Имя Отчество
2. Группа
3. Дата начала выполнения работы
4. Код работы
5. Название работы
6. Цели работы
7. Задачи работы

При необходимости при выполнении работы в отчет записываются контрольные значения.

Во время защиты тетрадь используется для вопросов преподавателя и ответов студента.

1. Работа ООП-301. Пользовательские шаблоны

Цели:
- изучение пользовательских шаблонов C++.
Задачи:
- исследование шаблонов функций;
- исследование шаблонов классов;
- исследование перегрузки шаблонов;
- разработка полезных шаблонов.

1.1. Подготовка проекта

Скачайте архив работы OOP100. Извлеките каталог OOP100 из архива в корневой каталог диска C:. Переименуйте каталог в OOP301. Откройте проект. Убедитесь, что целевой платформой является x86 или Win32.

Модуль main.cpp содержит основную функцию. Укажите в начале этого модуля сведения об организации, о себе, о проекте, тему и дату начала работы:

```
// ОТИ НИЯУ МИФИ
// 1ПО-00Д
// Студент Имя Отчество
// Объектно-ориентированное программирование
// ООР-301. Пользовательские шаблоны
// 01.01.2000
```

Модуль class.h предназначен для описания классов.

1.2. Шаблоны C++

Шаблон C++, — это *параметризованное относительно типа* определение функции или класса.

Иначе говоря, шаблон функции, — это функция, для которой дополнительно задается тип (типы), который будет использован как основной тип функции. Соответственно, шаблон класса, — это класс, для которого дополнительно задается тип (типы), который будет использован как основной тип класса.

Шаблоны формируются просто.

Сначала определяется функция или класс обычным образом.

Затем перед определением приписывается строка вида:

```
template <class T>
```

После этого в определении функции или класса название основного типа заменяется на `T`. Тип `T` становится параметром функции или класса.

Использовать класс-шаблон можно, только если указать тип параметра `T` при объявлении переменной.

Шаблон может иметь несколько параметров.

Параметры делятся на типированные и не типированные.

Типированный параметр, — это *тип*, который задается в приведенной выше строке при помощи ключевого слова class и *имени* типа.

Не типированный параметр, — это *константа*, которая задается при помощи типа константы и наименования, например:

```
template <int N>
```

Здесь шаблон имеет параметр, — константу N типа int.

Шаблоны C++, — очень мощный инструмент.

Условно все шаблоны можно поделить на те, которые пользователь создает сам (пользовательские) и те, которые поставляются вместе с языком C++ в виде так называемой *стандартной библиотеки шаблонов* STL (*Standard Template Library*). Шаблоны STL покрывают практически все потребности программистов в стандартных структурах данных, таких, как стеки, очереди или множества, в значительной мере облегчая программирование этих структур и реализацию алгоритмов. Именно поэтому мы должны изучать шаблоны как таковые и шаблоны STL в частности.

1.3. Шаблон функции

Рассмотрим функцию, которая меняет значения двух переменных.

Общепринятое название такой функции — swap.

Попробуем ее определить обычным образом.

Модуль class.h, описываем функцию swap для целого типа int:

```
swap(int & a, int & b) {
    int c = a;
    a = b;
    b = c;
}
```

Здесь используется принцип трех стаканов. Чтобы поменять содержимое двух стаканов с красным и белым вином, требуется третий стакан, иначе никак. Мы просто переливаем вино из одних стаканов в другие.

В основной функции main используем функцию swap для того, чтобы поменять местами значение двух переменных типа int:

```
int main(void) {
    int a = 1, b = 3;
    swap(a, b);
}
```

Заметим, что попытка обменять значения переменных другого типа не удастся. Попробуйте, например, заменить тип int на тип short.

Конечно, относительно легко использовать данную функцию для обмена значений других типов, используя обычное или статическое приведение. Но всякое приведение следует избегать, — один из полезных принципов программирования.

Поэтому лучше определить шаблон функции swap.

Для этого достаточно приписать перед функцией указанную выше строку, и заменить в теле функции тип int на тип T:

```
template <class T>
void swap(T & a, T & b) {
    T c = a;
    a = b;
    b = c;
}
```

Запускаем программу и видим, что она прекрасно работает. Для наглядности в окно просмотра Watch можно ввести переменные a и b.

Пробуем обменивать переменные других типов в main:

```
int main(void) {
    int a = 1, b = 3;
    swap(a, b);
    char x = 7, y = 9;
    swap(x, y);
    double p = 11, q = 13;
    swap(p, q);
}
```

Замечательно. Попробуем обменять переменные структурных типов.

Модуль class.h, описываем структурный тип:

```
struct point {
    int x, y;
    point(int a = 0, int b = 0)
        : x(a), y(b) {}
};
```

В основной функции пробуем обменять две точки:

```
int main(void) {
    int a = 1, b = 3;
    swap(a, b);
    char x = 7, y = 9;
    swap(x, y);
    double p = 11, q = 13;
    swap(p, q);
    point v(1, 1), w(2, 2);
    swap(v, w);
}
```

Опять все замечательно работает. Хороший шаблон получился.

Правда, я бы не рискнул обменивать две переменные типа класса строки с динамическим выделением памяти. Многое будет зависеть от того, как определен класс. Оставляю этот эксперимент для пытливых исследователей.

Однако попробуем разработать более интересный шаблон, а именно, шаблон функции сортировки.

Очистим функцию main.

Сначала нужно иметь собственно функцию сортировки. Для простоты используем bubble sort, то есть пузырьковую сортировку.

Описываем функцию в модуле class.h:

```
void bsort (int * a, int size) {
    for (int k = 1; k < size; k++) {
        for (int j = 1; j < size; j++) {
            if (a[j - 1] > a[j]) {
                swap(a[j - 1], a[j]);
            }
        }
    }
}
```

Проверяем, как работает сортировка, в функции main:

```
int main(void) {
    int a[] = { 5, 3, 1, 2, 4 };
    bsort(a, sizeof(a) / sizeof(int));
}
```

Массив типов int сортируется.

Несложно сделать из функции bsort шаблон, для чего нужно приписать перед функцией bsort строку:

```
template <class T>
```

после чего заменить тип int на тип T в первой строке функции только:

```
template <class T>
void bsort (T * a, int size) {
    . . .
}
```

В основной функции пробуем сортировать массив действительных чисел, например:

```
int main(void) {
    int a[] = { 5, 3, 1, 2, 4 };
    bsort(a, sizeof(a) / sizeof(int));
    double b[] = { 1.5, 1.3, 1.1, 1.2, 1.4 };
    bsort(b, sizeof(b) / sizeof(double));
}
```

Вот только как сортировать точки?

Нужно изменить шаблон таким образом, чтобы ему было известно, как сравнивать два объекта. Для этого добавляем в функцию шаблона третий параметр, указатель на функцию сравнения. В теле функции изменяем оператор сравнения, используя функцию сравнения cfunc:

```
template <class T>
void bsort (T * a, int size, int (*cfunc)(T, T)) {
    for (int k = 1; k < size; k++) {
        for (int j = 1; j < size; j++) {
            if (cfunc(a[j - 1], a[j])) {
                swap(a[j - 1], a[j]);
            }
        }
    }
}
```

Теперь при вызове функции нужно обязательно подставлять функцию сравнения объектов. Например, для типа int можно описать следующую функцию:

```
int compare_int(int a, int b) {
    return a < b;
}
```

Заметим, эта функция сортирует обратном порядке. Чтобы сделать сортировку в прямом порядке, нужно изменить знак операции отношения.

В основной функции снова сортируем массив целых чисел, используя функцию сравнения compare_int:

```
int main(void) {
    int a[] = { 5, 3, 1, 2, 4 };
    bsort(a, sizeof(a) / sizeof(int), compare_int);
    double b[] = { 1.5, 1.3, 1.1, 1.2, 1.4 };
    //bsort(b, sizeof(b) / sizeof(double));
}
```

Однако теперь нельзя сортировать действительные числа, для них нужна другая функция сравнения. Но можно попробовать сделать шаблон из функции сравнения, например, так:

```
template <class T>
int compare_number(T a, T b) {
    return a < b;
}
```

В основной функции применяем новую функцию сравнения:

```
int main(void) {
    int a[] = { 5, 3, 1, 2, 4 };
    bsort(a, sizeof(a) / sizeof(int), compare_number<int>);
    double b[] = { 1.5, 1.3, 1.1, 1.2, 1.4 };
    bsort(b, sizeof(b) / sizeof(double), compare_number<double>);
}
```

Для сортировки объектов типа point потребуется еще одна функция сравнения. Попробуйте разработать ее самостоятельно. Точки можно сортировать по расстоянию от центра координат, или же просто по сумме координат.

1.4. Шаблон класса массива

Для начала просто повторим шаблон класса Array из книжки. Он, кстати, может оказаться полезным, кроме того, он покажет нам, как обращаться с шаблонами классов.

Как уже говорилось, чтобы получить шаблон чего-нибудь, сначала нужно иметь либо функцию, либо класс, затем найти в них основные типы или константы и сделать их параметрами.

Класс Array предназначен для управления массивом целых чисел.

Числа для простоты хранятся в массиве размерности N.

N — константа, определенная в модуле class.h:

```
#define N 10
```

В модуле class.h описываем класс массива Array:

```
class Array {
    int a[N];
public:
    Array() {
        for (int i = 0; i < N; i++) a[i] = 0;
    }
    int & operator [] (int index) {
        assert(index >= 0 && index < N);
        return a[index];
    }
};
```

Здесь ничего необычного, операция индексирования возвращает ссылочный тип, который позволяет изменять элементы массива.

В основной функции можно использовать класс Array:

```
int main(void) {
    Array b;
    b[0] = 1;
    int x = b[0];
}
```

Отличия начинаются, когда мы сделаем шаблон. Основным типом T является int, кроме того, есть константа N типа int.

Модуль class.h, константу N комментируем.

```
//#define N 10

template <class T, int N>
class Array {
    T a[N];
public:
    Array() {
        for (int i = 0; i < N; i++) a[i] = 0;
    }
    T & operator [] (int index) {
        assert(index >= 0 && index < N);
        return a[index];
    }
};
```

Вносим в класс Array необходимые изменения.

В основной функции теперь нужно указать параметры:

```
int main(void) {
    Array<int, 10> b;
    b[0] = 1;
    int x = b[0];
}
```

Теперь шаблон может быть использован для произвольных типов и для любой размерности, не превышающей конструктивные ограничения.

Кроме этого, в него можно легко включить метод для сортировки.

1.5. Шаблон класса стека

Вторым полезным шаблоном является шаблон класса стека.

Опять же для простоты примем, что хранителем значений является простой массив. Здесь мы собираемся исследовать, как описываются методы шаблона вне класса.

Сначала заготовка класса стека, модуль class.h:

```
class tstack {
    int a[10];
    int count;
public:
    // конструктор
    tstack();
    // глубина стека
    int size(void);
    // проталкивает элемент
    int push(int element);
    // выталкивает элемент
    int pop(int & element);
    // элемент на вершине
    int top(int & element);
};

// конструктор
tstack::tstack() {
    count = 0;
    for (int i = 0; i < 10; i++) a[i] = 0;
}

// глубина стека
int tstack::size(void) {
    return count;
}

// проталкивает элемент
int tstack::push(int element) {
    return 1;
}

// выталкивает элемент
int tstack::pop(int & element) {
    return 1;
}

// элемент на вершине
int tstack::top(int & element) {
    return 1;
}
```

Не сомневаюсь, что эту заготовку вы самостоятельно доведете до кондиции. Методы, описанные вне класса, также должны начинаться с конструкции template, например:

```
template <class T>
int tstack<T>::size(void) { . . . }
```

Заметим, что методы проталкивания, выталкивания и элемента на вершине стека возвращают 1 в случае успеха операции, или 0, если операцию выполнить невозможно, например, стек пуст или, наоборот, полон.

После того, как все методы будут определены, нужно будет проверить работоспособность стека в функции main.

После отладки класса переходим к шаблону.

Заметим, что не все типы int являются основными, а только те, которые связаны с элементами стека. Вторым параметром шаблона является константа N типа int, как и в классе Array.

Операцию перевода класса в шаблон проделаете самостоятельно, дополнительно смотрите, как реализуются методы шаблона вне класса.

Тестировать шаблон нужно на типах int и point (обязательно!).

Кроме этого, нужно *обязательно* попробовать создать представителя шаблона класса tstack, используя генератор new.

1.6. Вопросы и упражнения

1. Что называется шаблоном функции, класса?
2. Что такое типированный параметр?
3. Что такое нетипированный параметр?
4. Как создается статический представитель шаблона класса?
5. Как создается динамический представитель шаблона класса?
6. Как описываются методы шаблонов классов вне класса?
7. В чем преимущества шаблонов?

2. Работа ООП-302. Стандартные шаблоны и алгоритмы

Цели:
- изучение библиотеки стандартных шаблонов;

Задачи:
- исследование шаблона vector (обязательное исследование);
- исследование шаблона list;
- исследование шаблона deque;
- исследование шаблона map (обязательное исследование);
- исследование стандартных алгоритмов;

2.1. Подготовка проекта

Скачайте архив работы OOP100. Извлеките каталог OOP100 из архива в корневой каталог диска C:. Переименуйте каталог в OOP302. Откройте проект. Убедитесь, что целевой платформой является x86 или Win32.

Модуль main.cpp содержит основную функцию. Укажите в начале этого модуля сведения об организации, о себе, о проекте, тему и дату начала работы:

```
// ОТИ НИЯУ МИФИ
// 1ПО-00Д
// Студент Имя Отчество
// Объектно-ориентированное программирование
// ООР-302. Стандартные шаблоны и алгоритмы
// 01.01.2000
```

Модуль class.h предназначен для описания классов.

2.2. Шаблоны STL

Большинство программ используют стандартные структуры данных, такие, как стеки, очереди, карты или множества. Для того, чтобы не создавать всякий раз одни и те же классы для работы со структурами данных, в язык C++ включена библиотека стандартных шаблонов STL (*Standard Template Library*). Цель данной работы, — исследование основных шаблонов, а также алгоритмов, которые могут быть применены к стандартным структурам данных.

Большинство шаблонов являются *контейнерами*. Контейнер — это объект, содержащий другие объекты. Контейнерные классы шаблонов STL делятся на *последовательные* и *ассоциативные* контейнеры.

Последовательные контейнеры предоставляют последовательный или произвольный доступ к элементам. Ассоциативные контейнеры оптимизированы для получения доступа к элементам по ключевым значениям.

2.3. Контейнер «Вектор»

Последовательные контейнеры STL — это вектор, список и дек. Соответствующие названия классов vector, list, deque.

Часто требуется организовать данные в виде массива, доступ к элементам которого производится по номеру элемента (индексу). Контейнер Вектор моделирует массив, но является более безопасным.

Для работы с вектором включаем модуль (модуль class.h):

```
#include <vector>
```

Шаблон вектора оптимизирован для доступа к элементам по индексу.

Вектор автоматически увеличивает свой размер, если первоначального размера оказалось недостаточно. Класс шаблона вектора определен следующим образом:

```
template <class T, class A = allocator<T>>
class vector {
    . . .
};
```

Первый параметр — это тип элементов вектора.

Второй параметр — это класс, отвечающий за выделение и освобождение памяти для элементов типа первого параметра. По умолчанию элементы добавляются при помощи оператора new и удаляются при помощи оператора delete. При создании элемента вызывается конструктор по умолчанию, поэтому, если в контейнер добавляются экземпляры класса, конструктор по умолчанию следует определить.

Для использования вектора удобно определить его тип при помощи оператора typedef. Например, если элементами вектора являются int, тип вектора определяется следующим оператором (модуль class.h):

```
typedef vector<int> intVector;
```

Перед использованием вектора его нужно создать (при помощи соответствующего конструктора). При этом вектору выделяется некоторое количество памяти. Примеры создания векторов:

```
int main(void) {
    intVector iv1;
    intVector iv2(10);
    intVector iv3(iv2);
}
```

В первом случае создается пустой вектор. Во втором случае создается вектор, в котором выделено 10 элементов. В третьем случае создается копия второго вектора. При этом следует учитывать, что копирование большого вектора займет определенное время.

2.3.1. Размер и объем

Следующие методы вектора управляют его размером и объемом.

Метод max_size возвращает максимальное количество элементов, которое может содержать вектор. Посмотрим, сколько элементов допускают векторы iv1 и iv2:

```
cout << "max_size = " << iv1.max_size() << endl;
cout << "max_size = " << iv2.max_size() << endl;
```

Запускаю программу на выполнение, изучаю результат. Жесть, даже не знаю, как произнести выведенное число 1 073 741 823, наверное миллиард. Ну да, памяти немало, четыре миллиарда байт, откуда только столько взялось, у меня оперативной памяти меньше. В любом случае это много.

Не факт, что у вас будет выведено то же число.

Метод capacity (*емкость*, *вместимость*) возвращает количество элементов, которое может уместиться в выделенной памяти.

Исследуем этот параметр:

```
cout << "capacity = " << iv1.capacity() << endl; // 0
cout << "capacity = " << iv2.capacity() << endl; // 10
```

Выводится 0 и 10.

Метод size возвращает актуальное количество элементов вектора, а метод empty возвращает истину, если size равно нулю. Заметим, что если capacity возвращает больше, чем size, значит, в вектор можно добавить capacity() - size() элементов без дополнительного выделения памяти:

```
cout << "size = " << iv1.size() << endl; // 0
cout << "size = " << iv2.size() << endl; // 10
```

Метод reserve(n) выделяет память для n элементов. При этом может произойти выделение дополнительной памяти. Попробуем:

```
iv1.reserve(10);
cout << "capacity = " << iv1.capacity() << endl; // 10
cout << "size     = " << iv1.size()     << endl; // 0
```

Метод resize(n) изменяет размер вектора до n элементов. При этом также может произойти выделение дополнительной памяти:

```
iv1.resize(20);
cout << "capacity = " << iv1.capacity() << endl; // 20
cout << "size     = " << iv1.size()     << endl; // 20
```

Следует также учитывать, что значения, возвращаемые этими методами, имеют тип vector<int>::size_type, поэтому нужно использовать приведение либо к типу int, либо наоборот.

2.3.2. Доступ к элементам

Доступ к элементам вектора может быть таким же, как и к элементам массива, при помощи перегруженной операции индексирования. Нужно только помнить, что эта операция, так же, как и для массивов, может привести к непредсказуемому результату, если заданный индекс превышает границы вектора, увы.

Есть, однако, метод at, который делает проверку индекса и в случае выхода за границу вектора выбрасывает исключение out_of_range.

Следующий цикл задает значения элементов вектора, используя операцию индексирования:

```
for (int i = 0; i < (int)iv2.size(); i++) {
    iv2[i] = i * 5;
}
```

Попробуем обратиться к элементу массива номер 100 при помощи метода at, убеждаемся, что исключение выбрасывается:

```
try {
    // выбрасывает исключение
    int x = iv2.at(100);
} catch(out_of_range) {
    cout << "Exception" << endl;
}
```

Методы front и back возвращают соответственно первый и последний элементы вектора:

```
cout << "front = " << iv2.front() << endl; // 0
cout << "back = "  << iv2.back()  << endl; // 45
```

2.3.3. Итераторы

Для работы с множеством элементов вектора (или любого другого контейнера) предусмотрены так называемые *итераторы*. Итератор оптимизирован для выборки элементов конкретного класса. Итераторы похожи на указатели в том смысле, что они ведут себя как указатели.

Итератор указывает на некоторый элемент контейнера. При прибавлении к итератору некоторого числа $\pm n$ итератор начинает указывать на элемент, отстоящий от текущего элемента на $\pm n$ позиций. Разыменование итератора возвращает элемент.

Функция begin устанавливает итератор на первый элемент контейнера, соответственно, функция end устанавливает итератор на последний элемент.

Обычный итератор перемещается по элементам в прямом направлении. Есть также *обратный* итератор, перебирающий элементы в обратном порядке.

Следующий пример показывает использование прямого итератора.

```
int j = 1;
intVector::iterator itor;
for (itor = iv2.begin(); itor != iv2.end(); ++itor) {
    *itor = j++;
    cout << "element = " << *itor << endl;
}
```

Итератор объявлен перед циклом. Переменная *j* используется для задания значений элементов.

Значение задается операцией разыменования итератора. Эта же операция используется для вывода значения.

Попробуем обратный итератор:

```
intVector::const_reverse_iterator rit;
for (rit = iv2.rbegin(); rit < iv2.rend(); ++rit) {
    cout << "element = " << *rit << endl;
}
```

Здесь следует обратить внимание на то, что для обратного итератора следует использовать другие функции, устанавливающие итератор на начало или конец, начинающиеся с *r*. Кроме того, и это важно, обратим внимание на условие завершения цикла. Для прямого итератора используется операция "!=", а для обратного, — операция "<". Заметим также, что функция end возвращает не последний элемент, а элемент, следующий за последним (которого нет).

Кроме того, для вывода элементов в обратном порядке был использован *константный* итератор, так как мы не изменяем значения элементов.

Теперь для вывода элементов вектора определим функцию showiv. Размещаем функцию в модуле class.h:

```
void showiv(intVector v) {
    cout << " size     = " << v.size()     << endl;
    cout << " capacity = " << v.capacity() << endl;
    int j = 0;
    intVector::const_iterator i;
    for (i = v.begin(); i != v.end(); i++) {
        cout << "element " << j++ << " = " << *i << endl;
    }
}
```

2.3.4. Модификаторы

Добавление или удаление элементов выполняют *модификаторы*.

Заметим, что добавление или удаление элементов могут приводить к большим временны*м* затратам, поскольку при этом потребуется перетасовка элементов и, возможно, перераспределение памяти.

В следующем примере в вектор вставляется один новый элемент:

```
cout << "\nadd one\n\n";
itor = iv2.insert(iv2.begin() + 5, 11);
cout << "current = " << *itor << endl; // 11
showiv(iv2);
```

Здесь метод insert возвращает указатель на добавленный элемент, который выводится, после чего выводятся все элементы вектора.

Следующий пример демонстрирует добавление нескольких элементов в тот же вектор (добавляется 5 значений 12):

```
cout << "\nadd five\n\n";
iv2.insert(iv2.begin() + 6, 5, 12);
showiv(iv2);
```

Здесь метод insert ничего не возвращает (объявлен как void).

Теперь проделаем обратные действия.

Сначала удалим пять элементов:

```
cout << "\nerase five\n\n";
iv2.erase(iv2.begin() + 6, iv2.begin() + 11);
showiv(iv2);
```

Обратим внимание на то, как задаются первый и последний удаляемый элементы, при помощи отсчета от начала.

Удалим один элемент:

```
cout << "\nerase one\n\n";
iv2.erase(iv2.begin() + 5);
showiv(iv2);
```

Попробуем добавление нескольких элементов из другого вектора.

Сначала зададим новый вектор iv4 со значениями, равными 50.

```
cout << "\ninsert from\n\n";
intVector iv4(10, 50);
iv2.insert(iv2.begin() + 5, iv4.begin(), iv4.begin() + 5);
showiv(iv2);
```

Здесь в методе insert сначала задается позиция в векторе, в который элементы вставляются, а затем диапазон элементов вектора, которые добавляются в первый вектор.

Наконец, используем еще три метода: push_back, pop_back, clear:

```
cout << "\npush-back\n\n";
iv2.push_back(100);
showiv(iv2);
cout << "\npop-back\n\n";
iv2.pop_back();
showiv(iv2);
cout << "\nclear\n\n";
iv2.clear();
showiv(iv2);
```

2.3.5. Операции

Библиотека STL определяет шесть перегруженных операций сравнения векторов, а также операцию обмена векторов (swap).

Функцию main скопируйте в модуль class.h, и переименуйте в vector1.

Сначала создадим два одинаковых вектора и сравним их:

```
int main(void) {
    intVector iv1(5), iv2(5);
    for (int i = 0; i < 5; i++) {
        iv1[i] = iv2[i] = i * 5;
    }
    showiv(iv1);
    showiv(iv2);
    if (iv1 == iv2) cout << "\nequal\n\n";
}
```

Вывод показывает, что векторы равны.

Заметим, что можно было бы заполнить значениями один вектор, а второй вектор приравнять первому операцией присваивания "=".

Сложнее с неравенством. Вектор 1 меньше вектора 2, если он либо меньше по размеру, либо меньше лексикографически (как строки).

Для проверки неравенства сначала удлиним вектор 2 и сравним:

```
iv2.push_back(25);
if (iv1 < iv2) cout << "\nv1 < v2\n\n";
```

Вывод показывает, что первый вектор меньше.

Теперь обменяем векторы, и заменим второй элемент первого вектора на ноль, при этом в первом векторе будет 6 элементов, а два первых элемента равны нулю. При этом этот вектор окажется меньше:

```
cout << "\nswap and compare\n\n";
iv1.swap(iv2);
iv1[1] = 0;
showiv(iv1);
showiv(iv2);
if (iv1 < iv2) cout << "\nv1 < v2\n\n";
```

Функцию main перенесем в class.h и переименуем в vector2.

2.4. Контейнер «Список»

Контейнер «Список» оптимизирован для произвольной вставки и удаления элементов, чем он и отличается от контейнера «Вектор», который оптимизирован для быстрого произвольного доступа.

Вследствие большей гибкости список обладает дополнительными операциями: splice, sort, merge, reverse и другими. Для работы со списком в class.h нужно включить модуль <list>, и определить тип:

```
#include <vector>
#include <list>

typedef vector<int> intVector;
typedef list<int> intList;
typedef list<int>::iterator lister;
```

Кроме того, для вывода списка определим в class.h функцию:

```
void showil(intList v) {
    cout << " size = " << v.size() << endl;
    int j = 0;
    intList::const_iterator i;
    for (i = v.begin(); i != v.end(); i++) {
        cout << "element " << j++ << " = " << *i << endl;
    }
}
```

Заметим, что у списка нет метода capacity, так как он не имеет смысла. Элементы списка выделяются динамически и не составляют непрерывный блок памяти, как у вектора. У списка нет операции индексирования и метода at, заполнение списка можно выполнить при помощи итератора:

```
int main(void) {
    intList L1(5), L2(5, 1);
    lister it;
    int i = 0;
    for (it = L1.begin(); it != L1.end(); it++) { *it = i++ * 5; }
    i = 0;
    for (it = L2.begin(); it != L2.end(); it++) { *it = i++; }
    showil(L1);
    showil(L2);
}
```

Операция splice похожа на один из вариантов insert:

```
cout << "\n splice\n\n";
L1.splice(++L1.begin(), L2, ++(++L2.begin()), --L2.end());
showil(L1);
showil(L2);
```

Здесь из второго списка вставляются элементы, с третьего по предпоследний, в первый список после первого элемента.

Поскольку для итератора этого контейнера не определена операция сложения с числами, приходится использовать операции инкремента и декремента. Получается сложнее, чем при операциях с вектором, но может это и не так важно. Заметим, что в отличие от вектора, операции списка работают с указателями, а не с элементами, что быстрее.

Как мы видели ранее, у вектора есть операции добавления элемента в конец и извлечения элемента с конца.

У списка есть аналогичные операции для добавления и удаления элемента с начала. Объясняется это тем, что вектор, — это непрерывный блок памяти, и добавление или удаление элемента с начала вызовет длительное перераспределение памяти. Тестируем добавление и удаление с начала:

```
cout << "\n front\n\n";
L1.pop_front();
L1.push_front(1);
showil(L1);
```

Убеждаемся, что удаление и добавление происходит. Заметим, что операции добавления и удаления с конца у списка также есть.

Рассмотрим теперь слияние списков, обращение и сортировку.

```
cout << "\n merge\n\n";
L1.merge(L2);
*(++(++L1.begin())) = 8;
showil(L1);
cout << "\n reverse\n\n";
L1.reverse();
showil(L1);
cout << "\n sort\n\n";
L1.sort();
showil(L1);
showil(L2);
```

После слияния списков в L1 второй и третий его элементы равны единице, поэтому я решил поменять третий элемент на значение 8. Как это

сделать, показано в третьей строчке. Переменная итератора не используется, поскольку begin возвращает итератор, и я просто разыменовываю его. Выглядит, наверное, сложно, но в таких конструкциях вся прелесть Си.

Заметим, что список L2 пуст.

Метод sort использует сравнение, определенное для типа списка.

Иногда же нужно использовать внешнюю функцию сравнения, как мы пробовали раньше в одной из работ, для этой цели есть соответствующий перегруженный метод sort(compare). При слиянии двух сортированных списков получается сортированный список:

```
cout << "\n merge sorted\n\n";
i = 0;
for (int i = 0; i < 5; i++) { L2.push_back(i); }
L1.merge(L2);
showil(L1);
```

Убеждаемся, что результирующий список сортирован.

2.4.1. Удаление элементов списка

Есть 4 метода для удаления элементов.

Метод remove(t) удаляет элемент, равный t.

Метод remove_if(predicate p) удаляет элементы, для которых выполняется предикат p.

Метод unique удаляет дублированные элементы. При этом дубликаты должны составлять непрерывную последовательность.

Метод unique(predicate p) удаляет дублированные элементы, для которых выполняется предикат p.

Скопируем функцию main в class.h и переименуем в list1.

Исследуем методы без предикатов:

```
int main(void) {
    intList L1(0);
    L1.push_back(1);
    L1.push_back(1);
    L1.push_back(1);
    L1.push_back(2);
    L1.push_back(3);
    L1.push_back(3);
    L1.push_back(1);
    L1.push_back(1);
    showil(L1);
    L1.remove(3);
    showil(L1);
    L1.unique();
    showil(L1);
}
```

Изначально в списке 8 элементов: 1, 1, 1, 2, 3, 3, 1, 1.

Метод remove удаляет элементы со значением 3.

Метод unique сокращает последовательности единиц до одной.

В результате в списке остается три элемента: 1, 2 и 1.

Для удаления с предикатом нужно определить его как функцию, желательно как шаблон. Модуль class.h:

```
// предикат
int pred(int i) {
    return (i > 1);
}
```

Используем данный предикат для удаления двойки из L1:

```
cout << "\n predicate\n\n";
L1.remove_if(pred);
showil(L1);
```

В результате в списке остается две единицы.

Функция предиката может быть сколь угодно сложной.

2.5. Контейнер «Дек»

Дек похож на вектор в том смысле, что он эффективен при последовательных чтении и записи. В отличие от вектора, он имеет эффективные операции проталкивания и выталкивания элемента как сзади, так и спереди. В этом смысле дек похож на список, он также использует выделение памяти отдельно для каждого элемента.

Для использования дека нужно сделать дополнительные объявления в модуле class.h:

```
#include <deque>
typedef deque<int> intDeque;
```

Функцию main перенесем в class.h под именем list2.

Исследуем проталкивание и выталкивание:

```
int main(void) {
    intDeque id;
    for (int i = 0; i < 5; i++) {
        id.push_front(i);
        id.push_back(i);
    }
    intDeque::iterator it;
    for (it = id.begin(); it != id.end(); it++) {
        cout << "element = " << *it << endl;
    }
    for (int i = 0; i < 5; i++) id.pop_front();
    cout << "\n";
    for (it = id.begin(); it != id.end(); it++) {
        cout << "element = " << *it << endl;
    }
}
```

Здесь в дек проталкиваются элементы одновременно слева и справа, после чего дек выводится на консоль. Затем выполняется выталкивание спереди пяти элементов и дек снова выводится на консоль.

Другие операции с деком аналогичны операциям вектора.

Кроме дека, STL определяет шаблон для стека stack с операциями: push, pop, top, size, empty, построенный на основе дека.

Другие последовательные контейнеры STL — очередь queue, приоритетная очередь priority_queue. Элементам приоритетной очереди приписываются значения приоритета.

2.6. Контейнер «Карта»

Ассоциативные контейнеры оптимизированы для быстрой выборки элементов по ключу, ассоциированному со значением. STL определяет шаблоны для ассоциативных контейнеров map (карта), multimap, set (множество) и multiset.

Для работы с контейнером map нужно сделать дополнительные изменения в модуле class.h:

```
#include <map>
#include <string>
```

Кроме этого, определим в модуле class.h класс для работы с картой:

```
class product {
    string _name;
    int _price;
public:
    product(string name = "", int price = 0) {
        _name = name;
        _price = price;
    }
    int price() {
        return _price;
    }
    int name() {
        return _name;
    }
};
```

Класс использует строковое значение в качестве названия продукта и целое значение в качестве цены.

Скопируем функцию main в модуль class.h и переименуем в deque.

Исследуем в функции main шаблон map:

```
int main(void) {
    product pen("pen", 10);
    product ball("ball", 20);
    product gum("gum", 15);
    map<string, product> pmap;
    pmap[pen.name()] = pen;
    pmap[ball.name()] = ball;
    pmap[gum.name()] = gum;
    cout << "size = " << pmap.size() << endl;
    cout << "pen price = " << pmap["pen"].price() << endl;
}
```

Здесь создается три предмета, затем они заносятся в карту по имени, объект pen извлекается по имени и выводится его цена.

Как видим, у карты есть метод size. Кроме него, есть методы max_size и empty. Определим в классе product метод для вывода в поток продукта:

```
class product {
    . . .
    friend ostream & operator<<(ostream & os, product p) {
        os << "product " << p.name()
           << "\tprice " << p.price() << endl;
        return os;
    }
};
```

Теперь в main используем итератор, чтобы вывести карту:

```
map<string, product>::iterator it;
for (it = pmap.begin(); it != pmap.end(); it++) {
    cout << it->first << "\t" << it->second;
}
```

Здесь используются свойства итератора first и second, указывающие соответственно на ключ и элемент.

Методы для поиска:

count(ключ) — возвращает количество элементов с данным ключом;

find(ключ) — возвращает элемент с данным ключом;

lower_bound(ключ) — возвращает элемент, ключ которого больше;

upper_bound(ключ) — возвращает элемент, ключ которого меньше.

Тестируем эти методы:

```
cout << "count pen = " << pmap.count("pen") << endl;
cout << "find pen  = " << pmap.find("pen")->second;
it = pmap.lower_bound("ball");
cout << "lower-bound ball = " << it->second;
it = pmap.upper_bound("ball");
cout << "upper-bound ball = " << it->second;
```

Заметим, что если элемент не найден, итератор принимает недопустимое значение и программа останавливается. Как проверить итератор, показывает следующий пример:

```
it = pmap.find("com");
if (it != pmap.end()) cout << "found\n";
```

Иначе говоря, если ничего не найдено, итератор указывает на конец.

Для добавления элемента в карту используются множественные методы insert, а для удаления — методы erase.

Следующий пример показывает вставку и удаление:

```
cout << "\n insert\n\n";
product table("table", 99);
pair<string, product> tab("table", table);
pair<map<string, product>::iterator, bool> p = pmap.insert(tab);
if (p.second) cout << "added table\n";
else cout << "insert failed\n";
pmap.erase("table");
```

Немного сложно. Коротко о картах все.

2.7. Стандартные алгоритмы

STL определяет также несколько стандартных алгоритмов:
for_each — для каждого элемента контейнера выполнить;
find — найти элемент с заданным значением;
find_if — найти элемент с использованием предиката;
count — подсчитать число элементов с заданным значением;
count_if — подсчитать число элементов по предикату.
Есть также алгоритмы для заполнения контейнеров значениями.
Функцию main скопируйте в class.h под именем map.
Исследуем только алгоритмы: for_each и find_if.
Сначала нужно включить в class.h модули и определить шаблоны классов печати и предиката для поиска:

```
#include <functional>
#include <algorithm>

// печать
template <class T>
class Print : public unary_function<T, void> {
public:
    void operator()(T& arg) {
        cout << arg << " ";
    }
};

// предикат
template <class T>
class Greater : public unary_function<T, int> {
public:
    int operator()(T& arg) {
        return (arg > 2);
    }
};
```

Наконец, в функции main описываем алгоритмы:

```
int main(void) {
    Print<int> print;
    vector<int> v(5);
    typedef vector<int>::iterator itor;
    for (int i = 0; i < 5; i++) { v[i] = i; }
    itor first = v.begin();
    itor last = v.end();
    cout << "\n for-each\n\n";
    for_each(first, last, print);
    cout << "\n\n find-if\n\n";
    Greater<int> preg;
    itor r = find_if(first, last, preg);
    if (r != last) cout << "find_if = " << *r << endl;
}
```

Особых пояснений, как мне кажется, здесь не требуется.
Остались неисследованными многие вопросы, например, итераторы, предикаты, не говоря уже о шаблонах и методах.

2.8. Вопросы и упражнения

1. Какой класс называется контейнерным?
2. Как классифицируются контейнерные классы?
3. Как оптимизируется класс вектора в сравнении с классом списка?
4. Какой тип возвращается методом size?
5. Что такое «итератор»?
6. Как осуществляется доступ к элементам вектора?
7. Как осуществляется доступ к элементам списка?
8. Как отследить попытку обращения к несуществующему элементу?
9. Что такое предикат? Как он используется?
10. Как проверить итератор на допустимость?
11. Как осуществляется доступ к элементам карты?
12. Как добавляются и удаляются элементы в/из вектора?
13. Как добавляются и удаляются элементы в/из списка?
14. Как добавляются и удаляются элементы в/из карты?

3. Работа ООП-303. Обработка исключений

Цели:
- изучение механизма исключений C++.

Задачи:
- изучение исключительных ситуаций;
- изучение механизма выброса и перехвата исключения;
- построение иерархии обработчиков;
- построение иерархии классов исключений.

3.1. Подготовка проекта

Скачайте архив работы OOP100. Извлеките каталог OOP100 из архива в корневой каталог диска C:. Переименуйте каталог в OOP303. Откройте проект. Убедитесь, что целевой платформой является x86 или Win32.

Модуль main.cpp содержит основную функцию. Укажите в начале этого модуля сведения об организации, о себе, о проекте, тему и дату начала работы:

```
// ОТИ НИЯУ МИФИ
// 1ПО-00Д
// Студент Имя Отчество
// Объектно-ориентированное программирование
// ООР-303. Обработка исключений
// 01.01.2000
```

Модуль class.h предназначен для описания классов.

3.2. Исключительные ситуации в программах

Исключительной называется такая ситуация в программе, которая исключает возможность ее дальнейшего нормального выполнения.

Первый классический пример исключительной ситуации связан с делением на ноль, которое может возникнуть, если в программе встречается операция деления. Это исключение аппаратное, возникает в процессоре и ведет к завершению выполнения программы, если программист явным образом не указал другое поведение.

Второй классический пример исключительной ситуации связан с файлами. Попытка открыть несуществующий файл для чтения встречается в программах независимо от квалификации программиста. Если программист не отслеживает факт открытия файла и создания дескриптора, рано или поздно программа также может быть остановлена. Эта ошибка не аппаратная, автоматически она не вызывает исключение, как деление на ноль, однако попытка чтения в файл, как правило, ведет в этом случае к программному исключению.

Аппаратные исключения возникают в аппаратуре и обрабатываются при помощи механизма структурированной обработки исключений.

Программные исключения генерируются программным кодом в случае обнаружения им исключительной ситуации. В рамках C++ эти исключения генерирует оператор throw.

3.3. Механизмы обработки исключений

При программировании в среде Win32 на языке C++ есть два механизма обработки исключений:

- структурированная обработка исключений;
- обработка исключений C++.

Структурированная обработка исключений не зависит от языка, поскольку в этом случае задействуется операционная система, однако она поддерживается языком C/C++, равно как и другими языками.

Две основные конструкции механизма структурированной обработки исключений следующие:

а) блок try-except, предназначен для обработки исключения.

б) блок try-finally, предназначен для выполнения очистки, например, высвобождения памяти, закрытия файлов и т.п. Структурированная обработка исключений изучается в курсе системного программирования.

Обработка исключений C++ более гибкая, потому что позволяет выбрасывать исключения разных типов, однако аппаратные исключения не могут быть перехвачены при помощи механизмов C++.

В C++ используется конструкция try-catch:

```
try {
    /* предположительно опасный код */
} catch(exception-type-1) {
    /* обработчик типа exception-type-1 */
} catch(exception-type-2) {
    /* обработчик типа exception-type-2 */
}
```

В принципе, этот механизм также является структурированным, и в значительной мере похожим на механизм операционной системы. Все современные объектно-ориентированные системы программирования поддерживают обработку исключений.

3.4. Обработка исключений C++

Как уже говорилось, обработка исключений средствами C++ отличается большей гибкостью, потому что выбрасывать и перехватывать можно любые типы данных. Блоков перехвата catch может быть несколько, а порядок блоков catch имеет значение. Последним блоком должен быть блок catch(…), в который попадают все не перехваченные исключения.

В пределах функции main построим блок try с множеством блоков catch, перехватывающих различные выбрасываемые типы.

Выбрасывать будем сами, при помощи оператора throw.

Примерный тестирующий код следующий:

```
int main(void) {
    int x = 3;
    short z = 5;
    try {
        throw 1;
    } catch(int x) {
        cout << "int thrown " << x << endl;
    } catch(double x) {
        cout << "double thrown " << x << endl;
    } catch(char x) {
        cout << "char thrown " << x << endl;
    } catch(char * x) {
        cout << "char * thrown " << x << endl;
    } catch(...) {
        cout << "... thrown\n";
    }
}
```

Теперь можно выбрасывать разные значения.

Последовательно подставляем в операторе throw значения:

```
1 x
1.0 x
x x
'x' x
"exception" x
z x
```

Убеждаемся, что срабатывают все обработчики.

Пробуем выполнить деление на ноль в блоке try:

```
try {
    int y = 0;
    x = x / y;
} catch . . .
```

Увы, обработчик catch(…) не перехватывает деление на ноль.

Если же мы попробуем заключить блок try в блок __try, возникнет ошибка: в одной функции нельзя одновременно использовать и структурированную обработку исключений, и обработку исключений C++.

Использовать механизм обработки исключений C++ можно каким угодно способом. Например, можно всегда выбрасывать целые числа, или константы, их обозначающие, каждое число при этом обозначает какую-то исключительную ситуацию.

Заметим, что исключения в этом случае выбрасывает программист, используя оператор throw, а исключительную ситуацию при этом нужно определить, используя обработку ошибок на месте.

Напомним, что обработка ошибки на месте, — это контроль значений в месте предположительного возникновения исключительной ситуации.

3.5. Обработка ошибок на месте

Пусть, например, нам нужно определить функцию, которая открывает некоторый файл для выполнения операций с ним.

Сначала опишем эту функцию, используя обработку ошибок на месте, используя стандартные механизмы. Нам нужен также файл. Для этого в каталоге C:\OOP303\Debug\ создайте текстовый файл 1.txt и запишите в него какой-нибудь текст, например, «Hello». В настройках проекта нужно также установить рабочий каталог "..\Debug".

В модуле class.h дополнительно включаем модуль:

```
#include <errno.h>
```

Модуль main.cpp. Описываем функцию, открывающую файл:

```
int open(LPCSTR path, FILE ** file) {
    *file = fopen(path, "rt");
    if (*file) return 1;
    if (errno == ENOENT) {
        cout << "No such file or directory.\n";
    } else if (errno == EACCES) {
        cout << "Access denied.\n";
    } else if (errno == EINVAL) {
        cout << "Empty or wrong path.\n";
    }
    return 0;
}
```

Здесь используется обработка ошибки на месте при помощи анализа значения глобальной переменной errno. Отслеживаемые значения:

- не существует указанного файла или каталога;
- доступ запрещен;
- неправильный аргумент функции fopen, например, пустой путь.

Код функции main простой:

```
int main(void) {
    FILE * file = 0;
    int result = open("", &file);
    if (!result) return;
    cout << "Success.\n";
}
```

Запускаем программу, убеждаемся, что выводится третье сообщение.

Теперь зададим неверный путь "11.txt".

При этом выводится первое сообщение.

Зададим верный путь "1.txt".

При этом выводится сообщение функции main. Если выводится первое сообщение, в настройках проекта нужно указать рабочий каталог.

Чтобы смоделировать ситуацию недоступности файла, его нужно создать администратором, установить права доступа только для администратора, и далее попытаться открыть, войдя в систему под ограниченной учетной записью.

3.6. Выбрасывание констант исключений

Теперь, когда понятно, как обнаруживаются ошибки, используем систему констант для выбрасывания исключений. Описываем новую функцию для открытия файла (main.cpp):

```
void openc(LPCSTR path, FILE ** file) {
    *file = fopen(path, "rt");
    cout << errno << endl;
    if (errno) {
        throw errno;
    }
}
```

В main выполняем разбор выброшенных значений:

```
int main(void) {
    FILE * file = 0;
    try {
        openc("", &file);
    } catch(int e) {
        if (e == ENOENT) {
            cout << "No such file or directory.\n";
            return;
        } else if (e == EACCES) {
            cout << "Access denied.\n";
            return;
        } else if (e = EINVAL) {
            cout << "Invalid arguments.\n";
            return;
        }
    }
    cout << "Success.\n";
}
```

Проверяем и убеждаемся, что обработка исключений выполняется.

3.7. Выбрасывание объектов

Наиболее удачным решением обработки исключений является формирование классов исключений. Для этого нужно описать иерархию классов исключений, и выбрасывать представителей этих классов.

Сначала нужен базовый класс. По идее он может быть абсолютно пустым. Однако мы снабдим его описанием ошибки.

Название базового класса file_error.

Модуль class.h:

```
struct file_error {
    string msg;
    file_error(LPCSTR m) : msg(m) {}
    LPCSTR message() {
        return msg.c_str();
    }
};
```

Далее описываем классы исключений для конкретных ошибок.

```
// класс ошибки неверный путь
struct fe_no_path : public file_error {
    fe_no_path() : file_error("No such file or directory") {}
};

// класс ошибки отказано в доступе
struct fe_access_denied : public file_error {
    fe_access_denied() : file_error("Access denied") {}
};

// класс ошибки неверные аргументы
struct fe_invalid_arg : public file_error {
    fe_invalid_arg() : file_error("Invalid arguments") {}
};
```

Таким образом, каждый объект класса содержит описание ошибки.

В модуле main.cpp описываем новую функцию для открытия файла:

```
void opencl(LPCSTR path, FILE ** file) {
    *file = fopen(path, "rt");
    cout << errno << endl;
    if (errno == ENOENT) {
        throw fe_no_path();
    } else if (errno == EACCES) {
        throw fe_access_denied();
    } else if (errno == EINVAL) {
        throw fe_invalid_arg();
    }
}
```

В основной функции разбираем выброшенный объект:

```
int main(void) {
    FILE * file = 0;
    try {
        open("", &file);
    } catch(fe_no_path) {
        cout << "No such file or directory.\n";
        return;
    } catch(fe_access_denied) {
        cout << "Access denied.\n";
        return;
    } catch(fe_invalid_arg) {
        cout << "Empty or wrong path.\n";
        return;
    }
    cout << "Success.\n";
}
```

Здесь мы используем свои сообщения потому, что не хотим принимать переменную исключения для большей ясности кода. Тестируем программу на разных значениях пути к файлу и убеждаемся, что все работает.

Напоследок в этой части используем сообщения объектов.

Следует обратить внимание, что если выкидывается статический объект (созданный без помощи new) и обработчик использует переменную, то параметр обработчика должен быть ссылкой, иначе будет создана копия объекта, что приведет к нарушению доступа к памяти.

Обработчик исключений использует полиморфизм.

Выбрасываемые представители классов являются производными от базового класса file_error. Следовательно, один-единственный блок catch может перехватывать все выбрасываемые объекты.

Новый вид функции main:

```
int main(void) {
    FILE * file = 0;
    try {
        opencl("", &file);
    } catch(file_error & e) {
        cout << e.message() << ".\n";
    }
}
```

Тестируем программу при разных значениях пути к файлу.

3.8. Иерархия классов исключений

Выбрасывание объектов имеет то преимущество, что можно построить иерархическую систему классов исключений, которая позволит отлавливать группы исключений, используя промежуточные базовые классы.

В качестве примера построим иерархию классов (модуль class.h):

```
class gen_err { virtual void foo() {} };
class file_err : public gen_err {};
class file_open_err : public file_err {};
class file_read_err : public file_err {};
class array_err : public gen_err {};
class array_bound_err : public array_err {};
class array_overflow_err : public array_err {};
```

Это двухуровневая система исключений, описывающая ошибки файлов, и ошибки массивов. В базовом классе gen_err есть виртуальная функция, единственная цель которой — включить механизм RTTI.

Для тестирования в функции main используем следующий код:

```
int main(void) {
    try {
        throw new file_open_err();
    } catch(array_err * e) {
        printf("catch 1: %s\n", typeid(*e).name());
    } catch(file_err * e) {
        printf("catch 2: %s\n", typeid(*e).name());
    } catch(gen_err * e) {
        printf("catch 3: %s\n", typeid(*e).name());
    } catch(...) {
        printf("catch all: Exception\n");
    }
}
```

Для тестирования исключений пробуем выбрасывать в операторе throw разные классы, все по очереди. При помощи механизма RTTI исключения можно классифицировать подробнее.

3.9. Исключения в конструкторах

Один из важных вопросов, связанных с исключениями, является вопрос об обнаружении ошибки в случае, если конструктор объекта выделяет динамическую память, а память не выделена по каким-то причинам.

Этот вопрос также следует исследовать. Следующий класс будет использован как выделяющий память. Модуль class.h:

```
struct m_class {
    char * a, * b;
    m_class(const char * a, const char * b) {
        a = new char[20];
        throw 1;
    }
};
```

В классе две переменных для выделения памяти. Это сделано намерено, чтобы исследовать вопрос, как избежать утечки памяти в случае, если память уже частично выделена.

Модуль main.cpp. В основной функции пробуем создать экземпляр:

```
int main(void) {
    try {
        m_class a("a", "b");
    } catch(int) {
        cout << "Exception.\n";
        return;
    }
    cout << "Success.\n";
}
```

Тестируем эту программу, убеждаемся, что выделение памяти переменной a происходит, исключение выбрасывается. При этом память для переменной a не освобождается.

Попробуем описать деструктор класса m_class.

```
~m_class() {
    cout << "in ~m_class()\n";
    if (a) delete[] a;
}
```

Снова тестируем, убеждаемся, что деструктор не срабатывает. Следовательно, освобождать память должен конструктор. Идея, которая может прийти на ум, — использовать блок try-finally. Модуль class.h, редактируем конструктор класса m_class:

```
m_class(const char * a, const char * b) {
    __try {
        a = new char[20];
        throw 1;
    } __finally {
        cout << "in finally\n";
        delete[] a;
    }
}
```

Тестируем этот вариант и убеждаемся, что блок finally выполняется, что и требовалось.

Есть, разумеется, и другие варианты, не использующие блок try. Например, убедившись, что память не выделена, освободить всю уже выделенную память, а потом выбрасывать исключение.

3.10. Вопросы и упражнения

1. Что называется исключительной ситуацией? Приведите примеры.
2. Назовите механизмы обработки исключений. Поясните различие.
3. В чем заключается различие между аппаратными и программными исключениями?
4. Что называется обработкой ошибок на месте? Приведите примеры.
5. Объясните обработку исключений при помощи иерархии классов исключений.
6. Измените конструктор класса m_class так, чтобы он определял исключения также для выделения памяти переменной b и освобождал эту память в случае исключения.