Лекция 12. Обработка исключений и введение в библиотеку STL

Казахский национальный университет имени аль-Фараби

Кафедра компьютерных наук

Объектно-ориентированное программирование

Лектор: старший преподаватель, PhD, Кенжебек Ержан Ғалымжанұлы

Цель лекции

Цель лекции

Изучить механизм обработки исключений C++ (throw, try, catch) и познакомиться с основами стандартной библиотеки шаблонов STL.

Ожидаемые результаты

1. Объяснять, зачем нужны исключения
2. Использовать конструкции throw, try и catch
3. Описывать основные сущности STL: контейнеры, алгоритмы, итераторы
4. Различать последовательные и ассоциативные контейнеры

План лекции

- Обработка исключений в программах
- Синтаксис исключений: throw, try, catch
- Библиотека STL

Исключения

- Исключения, второй большой вопрос данной лекции, позволяют применить систематический, объектно-ориентированный подход к обработке возникающих в классах C++ ошибок. Исключения — это ошибки, возникающие во время работы программы. Они могут быть вызваны множеством различных обстоятельств, таких, как выход за пределы памяти, ошибка открытия файла, попытка инициализировать объект недопустимым значением или использование индекса, выходящего за пределы вектора.

Для чего нужны исключения

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

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

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

Для чего нужны исключения

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

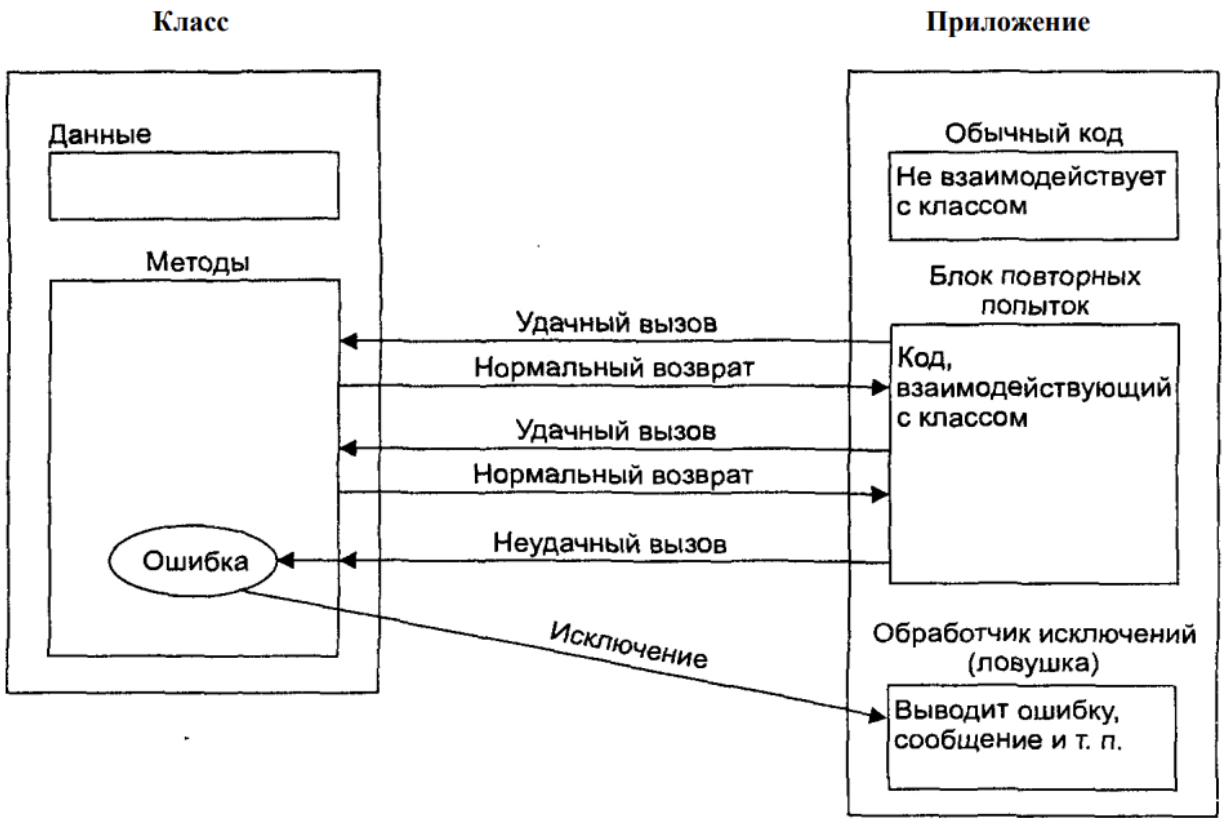
```
someClass obj1, obj2, obj3;
```

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

Введение в синтаксис исключений

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

Механизм исключений



Синтаксис исключений C++

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

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

Объяснение

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

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

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

Пример исключений (для класса Stack)

- Давайте, наконец, посмотрим на какую-нибудь реально работающую программу, использующую идею исключений. Пример, который вы сейчас увидите, является развитием программы **stack.cpp** из **предыдущей лекции**;
- в ней, если помните, создавалась стековая структура данных для хранения целочисленных значений. К сожалению, тот более ранний пример не обнаруживал двух важных ошибок. А именно: приложение могло пытаться извлечь или поместить в стек слишком много объектов, что привело бы либо к **получению неправильных данных**, либо к **превышению размера массива**. В программе **xstack.cpp** мы используем механизм исключений, чтобы обрабатывать эти ситуации.

Пример исключений (код)

```cpp
// xstak.cpp
// Демонстрация механизма исключений
#include <iostream>
using namespace std;
const int MAX = 3; // в стеке максимум 3 целых числа
//////////////////////////////////////////////////////////////
class Stack
{
private:
    int st[MAX]; // стек: целочисленный массив
    int top; // индекс вершины стека
public:
    class Range // класс исключений для Stack
    {           // внимание: тело класса пусто
    };
    //--------------------------------------------------------
    Stack() // конструктор
    {
        top = -1;
    }
    //--------------------------------------------------------
    void push(int var)
    {
        if (top >= MAX - 1) // если стек заполнен,
            throw Range(); // генерировать исключение
        st[++top] = var; // внести число в стек
    }
    //--------------------------------------------------------
    int pop()
    {
        if (top < 0) // если стек пуст,
            throw Range(); // исключение
        return st[top--]; // взять число из стека
    }
};
//////////////////////////////////////////////////////////////
int main()
{
    Stack s1;
    try
    {
        s1.push(11);
        s1.push(22);
        s1.push(33);
        // s1.push(44); // Опаньки! Стек заполнен
        cout << "1: " << s1.pop() << endl;
        cout << "2: " << s1.pop() << endl;
        cout << "3: " << s1.pop() << endl;
        cout << "4: " << s1.pop() << endl; // Опаньки! Стек пуст

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



xstack.cpp

Разбор кода

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

```
class Range
  {    // Тело класса пусто!
  };
```

- В программе xstack.cpp исключение генерируется в двух местах, оба раза с помощью выражения:

```
throw Range();
```

- Блок повторных попыток (**try** – блок). Все выражения в main(), в которых могут произойти ошибки, то есть выражения, манипулирующие данными объектов класса Stack, заключены в фигурные скобки, перед которыми стоит ключевое слово try:

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

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

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

Summary: throw, try, catch

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

STL – Стандартная библиотека шаблонов

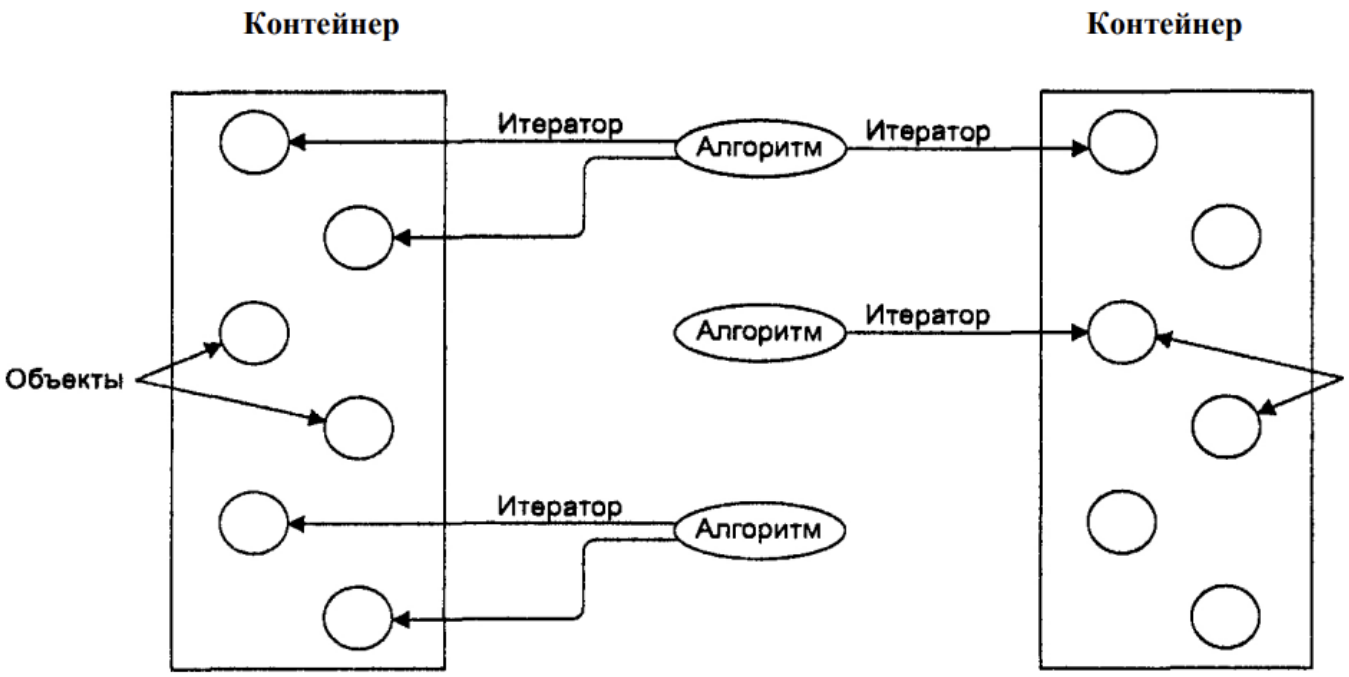
- В университетские учебные планы по специальности «Информатика» обычно входит курс «Структуры данных и алгоритмы». Термин структура данных говорит о том, как хранится информация в памяти компьютера, а алгоритм — как эта информация обрабатывается.

- Классы C++ предоставляют прекрасный механизм для создания библиотеки структур данных. В прошлом производители компиляторов и разные сторонние разработчики ПО предлагали на рынке библиотеки классов-контейнеров для хранения и обработки данных. Теперь же в стандарт C++ входит собственная встроенная библиотека классов-контейнеров. Она называется Стандартной библиотекой шаблонов (в дальнейшем мы будем употреблять сокращение STL) и разработана Александром Степановым и Менг Ли из фирмы Hewlett Packard. STL — это часть Стандартной библиотеки классов C++, которая может использоваться в качестве этакой повседневным спасателем для хранения и обработки данных.

Введение в STL

- В STL содержится несколько основных сущностей. Три наиболее важные из них — это контейнеры, алгоритмы и итераторы.
- **Контейнер** — это способ организации хранения данных. В предыдущих главах нам уже встречались некоторые контейнеры, такие, как стек, связный список, очередь. Еще один контейнер — это массив, но он настолько тривиален и популярен, что встроен в C++ и большинство других языков программирования. Контейнеры бывают самые разнообразные, и в STL включены наиболее полезные из них.
- Под **алгоритмами** в STL подразумевают процедуры, применяемые к контейнерам для обработки их данных различными способами. Например, есть алгоритмы сортировки, копирования, поиска и объединения. Алгоритмы представлены в STL в виде шаблонных функций. Однако они не являются методами классов-контейнеров. Наоборот, это совершенно независимые функции. На самом деле, одной из самых привлекательных черт STL является универсальность ее алгоритмов.
- Итераторы — это обобщение концепции указателей: они ссылаются на элементы контейнера. Их можно инкрементировать, как обычные указатели, и они будут ссылаться последовательно на все элементы контейнера. Итераторы — ключевая часть всего STL, поскольку они связывают алгоритмы с контейнерами. Их можно представить себе в виде кабеля, связывающего колонки вашей стереосистемы или компьютер с его периферией.

Контейнеры



Алгоритмы используют итераторы для работы с объектами контейнеров.

STL включает в себя семь основных типов контейнеров и еще три производных типа. Зачем же нам столько типов контейнеров? ***Почему нельзя использовать обычный массив во всех случаях, когда нужно хранить данные***? Ответ таков: **неэффективно**. Работать с массивом во многих ситуациях бывает неудобно — медленно и вообще затруднительно.

Виды контейнеров

- Контейнеры STL подразделяются на две категории:
  - последовательные
  - ассоциативные.
- Среди последовательных выделяют **векторы**, **списки** и **очереди с двусторонним доступом**.
- Среди ассоциативных — **множества (set), мультимножества (multiset), отображения (map) и мультиотображения (multimap)**.
- Кроме того, наследниками последовательных выступают еще несколько специализированных контейнеров: **стек, очередь и приоритетная очередь.**

Последовательные контейнеры

Контейнер	Характеристика	Плюсы/минусы
Обычный массив	Фиксированный размер	Скоростной случайный доступ (по индексу) Медленная вставка или изъятие данных из середины Размер не может быть изменен во время работы программы
Вектор	Перераспределяемый, расширяемый массив	Скоростной случайный доступ (по индексу) Медленная вставка или изъятие данных из середины Скоростная вставка или изъятие данных из хвоста
Список	Аналогичен связному списку	Скоростная вставка или изъятие данных из любого места Быстрый доступ к обоим концам Медленный случайный доступ
Очередь с двусторонним доступом	Как вектор, но доступ с обоих концов	Скоростной случайный доступ Медленная вставка или изъятие данных из середины Скоростная вставка и изъятие данных из хвоста или головы

STL применение

- **Продолжение будет в следующей лекции 13:**
  - **Ассоциативные контейнеры**
  - **Методы контейнеров**
  - **Адаптеры контейнеров**
  - **Алгоритмы**
  - **Итераторы**
  - **Пример в задаче**

Выводы

1. Исключения — систематический способ обработки ошибок, избавляющий от проверки кода возврата после каждого вызова.
2. Исключение генерируется оператором throw, потенциально опасный код помещается в блок try, обработка выполняется в блоке catch.
3. Исключения позволяют сообщать об ошибках из конструкторов, которые не возвращают значения.
4. STL основана на трёх сущностях: контейнеры, алгоритмы и итераторы.
5. Контейнеры делятся на последовательные (vector, list, deque) и ассоциативные (set, multiset, map, multimap).

Контрольные вопросы

1. Почему исключения удобнее проверки кодов возврата?
2. Как работают операторы throw, try и catch?
3. Что произойдёт, если исключение не перехвачено ни одним обработчиком?
4. Как использовать исключения для сообщения об ошибке в конструкторе класса?
5. Что такое STL и из каких основных сущностей она состоит?
6. Чем последовательные контейнеры отличаются от ассоциативных? Приведите примеры.

Список литературы

Основная литература

1. Лафоре Р. Объектно-ориентированное программирование в C++. 4-е изд. — СПб.: Питер, 2004.
2. Страуструп Б. Язык программирования C++. Специальное издание. — М.: Бином, 2011.
3. Джосаттис Н. Стандартная библиотека C++: справочное руководство. 2-е изд. — М.: Вильямс, 2014.

Дополнительная литература

4. Липпман С., Лажойе Ж., Му Б. Язык программирования C++. Базовый курс. 5-е изд. — М.: Вильямс, 2014.
5. Шилдт Г. Полный справочник по C++. 4-е изд. — М.: Вильямс, 2006.
6. Stroustrup B. A Tour of C++. 3rd ed. — Addison-Wesley, 2022.

Электронные ресурсы

7. Справочник по C++ и стандартной библиотеке: en.cppreference.com
8. C++ Core Guidelines: isocpp.github.io/CppCoreGuidelines

- Спасибо за внимание!