Тема: Обработка исключительных ситуаций в C++

- Проблема обработки ошибок
- Идея обработки исключений в C++
- Реализация обработки исключений
- Раскручивание стека вызовов
- Особенности обработки исключений в C++
- Классы исключений
- Группировка и иерархия исключений

Сложность задачи обработки ошибочных ситуаций

- Концепция программирования с защитой от ошибок
- Что считать ошибкой?
- «Полицейский» принцип обработки ошибочных ситуаций
- Возврат признака завершения
- «Регистрация» ошибок (значение глобальной переменной ***errno*** в C)
- Обработчики ошибочных ситуаций

Проблема обработки ошибок

Главная проблема обработки исключительных ситуаций состоит в том, что

- **разработчику библиотеки классов просто обнаружить возникновение ошибки при выполнении программы, но сложно принять решение, как с ней поступить.**
- **пользователь знает, что предпринять в случае ошибки, но не имеет возможности ее обнаружить (иначе ошибка была бы обработана в пользовательском коде, и от библиотеки ничего бы не потребовалось)**

Идея обработки исключений в C++

- Пространственное разделение события возникновения исключения и его обработки
- Понятие исключения, как объекта и обработчика исключения, как функционального блока, принимающего аргумент
- Механизм поиска обработчика путем разматывания стека
- Предопределенная обработка исключений

Пример заявления исключения типа **const char***

```
class String {
    public:
     // ..........................
     char& operator[ ] ( int ){
         if( i < 0 || length() < i )
      throw "Index out of range.";
         return text[i];
     };
     // ..........................
};
```

Использование кода прячем в **try-**блок

```
void foo( String& s ) {
    // . . .
    try {
      do_something ( s );
    }
    catch ( const char* s ) {
      // здесь пользователь имеет возможность
      // обработать исключительную ситуацию
      // например, так
      cerr << “Произошла ошибка: ” << s << “\n”;
      exit(1);
    }
    // Здесь продолжается выполнение при нормальном
    // завершении функции do_something()
}
```

Синтаксические элементы и правила

- Слова **throw**, **try** и **catch** являются ключевыми словами языка C++
- Конструкция **catch (. . . . .) { . . . . . }** называется ***обработчиком исключения*** и должна размещаться сразу за **try**-блоком, либо за другим обработчиком.
- В круглых скобках содержится имя типа и, возможно, имя аргумента
- Аргумент может использоваться подобно аргументу функции и передавать обработчику дополнительную информацию, связанную с исключением – значение исключения.

Раскручивание стека вызовов

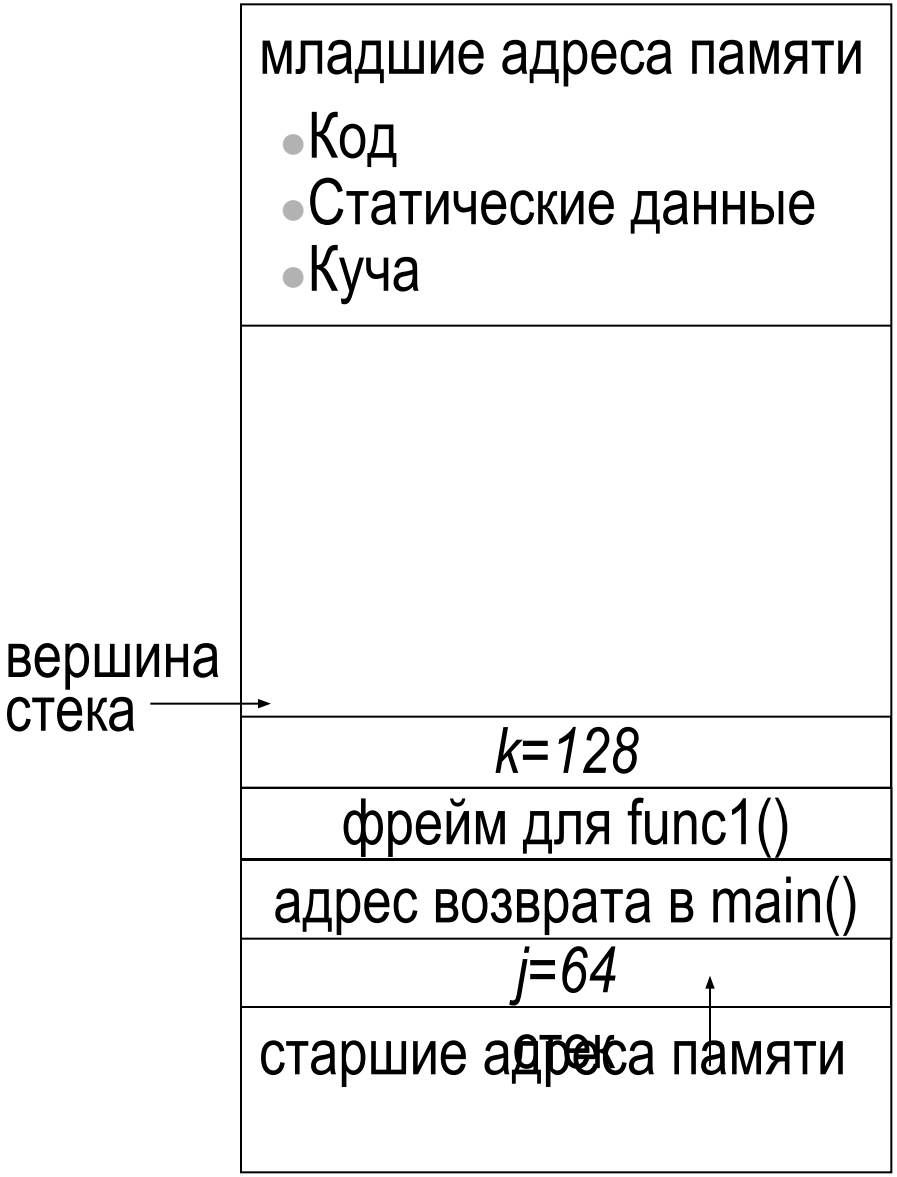
- Процесс генерации и последующей обработки исключений требует поиска соответствующего обработчика от точки возникновения исключения через последовательность стековых фреймов вызовов функций до фрейма функции, имеющей искомый обработчик. Говорят, что происходит ***раскручивание стека вызовов***.

Изменение стека вызовов – I

```
void func2( ) {
    int  i=256;
}

void func1( ) {
    int  k=128;
    func2();
}
void main( void ){
    int  j=64;
    func1();
}
```

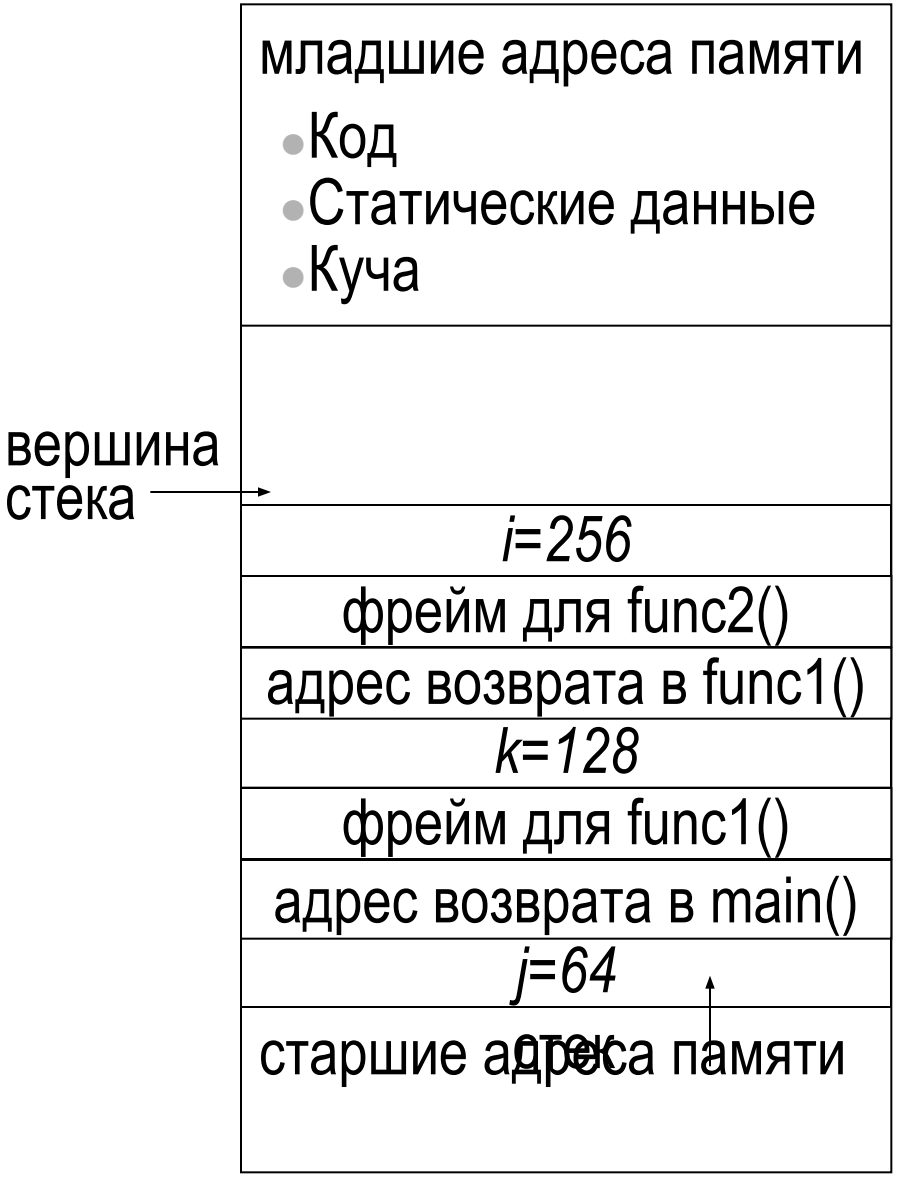
Изменение стека вызовов – II

```
void func2( ) {
    int  i=256;
}

void func1( ) {
    int  k=128;
    func2();
}
void main( void ){
    int  j=64;
    func1();
}
```



Изменение стека вызовов – III

```
void func2( ) {
    int  i=256;
}

void func1( ) {
    int  k=128;
    func2();
}
void main( void ){
    int  j=64;
    func1();
}
```



Механизмы раскручивания программного стека

- Естественное, в процессе выполнения программы;
- Связанное с обработкой исключительных ситуаций;
- Путем применения функции **longjmp()**;
- Путем прямой модификации регистра указателя стека CPU

Особенности обработки исключений в языке C++

- Обрабатываются только исключения, явно генерируемые некоторой функцией;
- Поддерживается ***окончательная модель обработки*** (т.е. после возникновения исключения невозможно продолжение выполнения программы от точки исключения);
- Обработка исключения возможна только в функции, вызванной до его появления и еще не завершившейся. После выбрасывания исключения управление должно быть передано некоторому программному блоку, принадлежащему функции, еще находящейся в стеке вызовов, путем его разматывания;

- Если заявлено исключение, для которого нет обработчика в цепочке вызовов, программа будет завершена. В процессе поиска обработчика программный стек будет раскручен до конца;
- Если обработчик «поймал» исключение, то обработка этого же исключения другими обработчиками, которые могут для него существовать, невозможна. (Другими словами, действует первый подходящий обработчик, встретившийся в процессе разматывания стека);

- Если после заявления исключения управление передано **catch**-блоку, то вне зависимости от результата последующих действий исключение считается обработанным;
- Обработчик, как и обычная функция, может заявить исключение. Более того, в нем может использоваться оператор **throw** без параметра, что означает повторное генерирование исключения, обрабатываемого в данный момент;

Обработка исключений и традиционные способы защиты

- «Полицейский» подход
  - можно быть и поумнее
  - соответствует случаю отсутствия обработчика
- Возврат значения-сигнала ошибки
  - не всегда возможно (например, любой символ для операции индексации строки допустим)
  - проверка утомительна, удлиняет и загромождает код

- Нормальное завершение с установкой внешнего признака ошибки (***errno***)
  - ошибка м.б. незамечена
  - сколько признаков надо иметь?
- Вызвать функцию-обработчик в случае ошибки (error handler function)
  - близок к механизму обработки исключений
  - нет единого стандарта
  - проблема программирования функций-обработчиков «по умолчанию»

Что такое исключение?

- Исключение выступает одновременно как переменная и как тип данных (или как объект и как класс)
- Оператор **throw** выбрасывает объект, а **catch**-обработчик ловит класс (или **throw** выбрасывает переменную, а **catch** ловит ее тип)

Генерирование исключений

При генерации исключения (**throw**) функции библиотеки исполняющей системы осуществляют следующие действия

- Создается и запоминается ***копия*** объекта (переменной). Т.е. если в точке генерации исключения оказывается недоступен копирующий конструктор (например, он не является **public**, а исключение заявляет не функция-друг), то возникает сообщение об ошибке.
- Разматывается стек, с вызовом деструкторов локальных объектов, выходящих из области видимости

Распознавание исключений

- Управление передается ближайшему **catch**-обработчику, совместимому с типом выброшенного исключения
- При этом копия объекта-исключения передается, если это предусмотрено, обработчику в качестве параметра.

Обработка исключений

- Обработчик считается найденным, а исключение обработанным, если:
  - тип исключения соответствует типу, ожидаемому в обработчике. Переменной (объекту) типа **T** соответствует обработчик, перехватывающий **T, const T, T&, const T&**
  - тип обработчика является публичным базовым классом для заявленного исключения
  - обработчик ожидает указатель, и исключение является указателем, который может быть преобразован к типу обработчика по стандартным правилам преобразования указателей
  - встретился обработчик по умолчанию

Обработчик по умолчанию

- Обработчик по умолчанию вызывается для исключения любого типа и имеет вид

```
catch( ... ) {
    // тело обработчика
}
```

- Обработчик по умолчанию должен располагаться последним среди обработчиков данного **try**-блока.

Пример

- Пусть класс ***Vector*** должен обнаруживать ошибки индексации и выделения памяти

```
class Vector {
    public:
        class Range{};
        class Memory{};
            // . . . . .
        Vector(int);
        int& operator[ ](int);
            // . . . . .
    private:
        int lbound, ubound;
        int *v;
};
```

Пример (продолжение)

```
Vector::Vector (int size) {
    if( !(v=new int[size]) ) throw Memory();
    else { lbound = 0; ubound = size – 1; }
}
int& Vector::operator[](int i) {
    if( i < lbound || ubound < i ) throw Range();
    else return v[i – lbound];
}
```

Пример (продолжение)

- Пользователь класса ***Vector*** может различить два исключения, включив два обработчика:

```
void f(void) {
    try{
        use_vectors();
    }
    catch( Vector::Range ){
      // тело обработчика ошибки индексации
    }
    catch( Vector::Memory ){
      // тело обработчика ошибки выделения памяти
    }
}
```

Повторное заявление исключений

- Исключение считается обработанным сразу после входа в обработчик. Поэтому

```
try{
    do_something();
}
catch( ExeptionType ){
    // . . . . .
    throw ExeptionType();
}
```

не вызовет бесконечного цикла.

Вложенные обработчики

```
try {
      // . . . . .
}
catch( type ) {
       try {
           // код, в котором возможно
           // возникновение ситуации типа
       type
       }
       catch( type ) {
           // . . . . .
       }
}
```

не вызовет бесконечного цикла

Тип и значение исключений

- При выходе индекса за границы можно передать ошибочное значение

```
class Vector {
    public:
    class Range{
        public:
        int  index;
        Range( int i ) : index( i ) { };
    };
                // . . . . .
        int& operator[ ](int);
                // . . . . .
    private:
            int lbound, ubound;
            int *v;
};
```

```
int& Vector::operator[ ](int i) {
    if( i < lbound || ubound < i ) throw Range(i);
    else return v[i-lbound];
}
void f( Vector& v ) {
    // . . . . .
    try{
        use_vectors(v);
    }
    catch( Vector::Range r ){
        cerr << “Bad index: “ << r.index << ‘\n’;
        // . . . . .
    }
    // . . . . .
};
```

Группировка исключений

```
enum MathError { Overflow, Underflow, Zerodivide, /*. . . */ };
    // . . . . .
try {
    // . . . . .
    MathError m;
    throw m;
}
catch( MathError m ) {
    switch (m) {
        case Overflow :    /* . . . . . */   break;
        case Underflow :  /* . . . . . */   break;
        case Zerodivide :  /* . . . . . */   break;
        default : /* . . . . . */;
    }
}
```

Иерархия исключений

- Объектно-ориентированный подход дает лучшее решение

```
class MathError { };

class Overflow  :  public MathError { };

class Underflow : public MathError { };

class Zerodivide : public MathError { };

// . . . . .
```

Далее…

```
try {
    // . . . . .
}
catch( Overflow ) {
    // обработка Overflow и его потомков
}
catch( MathError ) {
    // обработка любого MathError,
    // кроме Overflow и его потомков
}
```

РЕЗЮМЕ

- Задача обработки ошибочных ситуаций сложна и есть разные подходы
- C++ предлагает свой механизм обработки исключений
- Для создания кода, обрабатывающего исключения, имеются стандартные приемы
- Используйте классы и объекты для исключений
- Используйте иерархию классов для группировки исключений