# ППОИС

## Часть 1

Обеспечение отказоустойчивости программных систем

# Что такое исключение?

Программа может быть правильной. Но при ее работе все равно могут возникать неприятные ситуации, например, отсутствие достаточного объема памяти, попытка чтения испорченного файла с диска, переполнение при арифметических операциях и многие другие.

Относительно недавно в компиляторах языка **C++** появились рекомендованные стандартом ANSI средства для обработки особых ситуаций. Такие ситуации в C++ называют ***исключительными ситуациями или исключениями*** **(Exceptions)**.

Механизм обработки особых ситуаций присутствовал в разных языках программирования до появления **C++**.

В языке **C++** практически любое состояние, достигнутое в процессе выполнения программы, можно заранее определить как особую ситуацию (исключение) и предусмотреть действия, которые нужно выполнить при ее возникновении.

# Общая схема обработки исключений

Базовый принцип, на котором основана обработка исключений, — восстановление состояния и выбор альтернативных действий в случае ошибки. Предположим, в вашей программе имеется некий блок и вы не уверены, что он доработает до конца. При выполнении блока может возникнуть нехватка памяти, или начнутся проблемы с коммуникациями, или нехороший клиентский объект передаст неверный параметр. Разве не хотелось бы написать программу в таком виде:

```
if (блок будет работать) {
    блок;
}
else {
  сделать что-то другое;
}
```

С помощью исключений вы «допрашиваете» подозрительный блок. Если в нем обнаружится ошибка, компилятор поможет восстановить состояние перед выполнением блока и продолжить работу.

# Новые операторы языка C++

Для реализации механизма обработки исключений в язык C++ введены следующие три ключевых (служебных) слова:

- **try** (контролировать),
- **catch** (ловить),
- **throw** (генерировать, порождать, бросать, посылать, формировать).

```
…
try
{ …
  if (b==0) throw ”Ошибка!”;
  b = a/b;
  …
}
catch(char * str)
{
  cout << ”При выполнении: ”
  << str;
}
 …
```

# Операторы обработки исключений

Служебное слово **try** позволяет выделить в любом месте исполняемого текста программы так называемый контролируемый блок:

**try { <операторы> }**

Среди операторов, заключенных в фигурные скобки могут быть описания, определения, обычные операторы языка C++ и специальные операторы генерации (порождения, формирования) исключений:

**throw <выражение_генерации_исключения>;**

Когда выполняется такой оператор, то с помощью выражения, использованного после служебного слова **throw**, формируется специальный объект, называемый ***исключением***.

# Операторы обработки исключений 2

Исключение создается как статический объект, тип которого определяется типом значения ***выражения_генерации_исключения***.

После формирования исключения исполняемый оператор **throw** автоматически передает управление (и само исключение как объект) непосредственно за пределы контролируемого блока.

В этом месте (за закрывающейся фигурной скобкой) обязательно находятся один или несколько обработчиков исключений, каждый из которых идентифицируется служебным словом **catch** и имеет в общем случае следующий формат:

**catch (тип_исключения имя) { <операторы> }**

# Блок обработки исключений

Об операторах в фигурных скобках за ключевым словом **catch** говорят как о ***блоке обработчика исключений***. Обработчик исключений (процедура обработки исключений) внешне и по смыслу похож на определение функции с одним параметром, не возвращающей никакого значения.

Когда обработчиков несколько, они *должны отличаться друг от друга типами исключений*. Все это очень похоже на **перегрузку функций**, когда несколько одноименных функций отличаются спецификациями параметров. Так как исключение передается как объект определенного типа, то именно этот тип позволяет выбрать из нескольких обработчиков соответствующий посланному исключению.

# Общий синтаксис перехвата исключений

Чтобы перехватить исключение, программа должна представлять собой следующую конструкцию:

```
try {
    // Фрагмент, который может инициировать исключения
}

catch (Exception_Type t) {
    // Восстановление после исключения типа Exception_Type
}

catch (...) {
    // Восстановление после исключений всех остальных типов
}
```

# Синтаксис инициирования исключений

Рассмотрим следующий пример:

```
enum Bounds { kLow, kHigh };
void fn(int x) throw(Bounds) {
    if (x < 0)
        throw kLow;  // Функция завершается здесь
    if (x > 1000)
        throw kHigh; // Или здесь
// Теперь делаем то, что надо
}
```

В 1-й строке определяется тип исключения. Исключения могут иметь любой тип: целое, перечисление, структура, класс. Во 2-й строке объявляется интерфейс функции с новым придатком — спецификацией исключений, который определяет, какие исключения могут быть получены от функции вызывающей стороной. В данном примере инициируется исключение единственного типа Bounds. В 4-й и 6-й строке показано, как инициируются исключения, которые должны быть экземплярами одного из типов, указанного в спецификации исключений данной функции.

# Спецификации исключений

*Спецификации исключений должны подчиняться следующим правилам:*

## Объявления и определения

Спецификация исключений в объявлении функции должна **точно совпадать** со спецификацией в ее определении.

```
void Fn() throw(int); // Объявление


    // Где-то в файле .cpp …

void Fn()  throw(int) // Реализация
{  . . .  }
```

Если определение будет отличаться от объявления, компилятор откажется компилировать определение.

# Спецификации исключений

*Спецификации исключений должны подчиняться следующим правилам:*

## Функции без спецификации исключений

Если функция не имеет спецификации исключений, она может инициировать любые исключения. Например, следующая функция может инициировать что угодно и когда угодно.

**void fn();** // Может инициировать исключения любого типа

## Функции, не инициирующие исключений

Если список типов в спецификации пуст, функция не может инициировать никакие исключения.

Разумеется, при хорошем стиле программирования эту форму следует использовать всюду, где вы хотите заверить вызывающую сторону в отсутствии инициируемых исключений.

**void fn() throw();** // Не инициирует исключений

# Спецификации исключений

*Спецификации исключений должны подчиняться следующим правилам:*

**void fn() throw();** // Не инициирует исключений

Эта строка эквивалентна использованию атрибута __declspec (nothrow). Его использование не является обязательным.

(**C++11**) В стандарте ISO C++11 был представлен оператор noexcept. Он поддерживается в Visual Studio 2015 и более поздних версий.

Когда это возможно, используйте **noexcept**, чтобы задать возможность вызова функцией исключений.

# Спецификации исключений

*Visual C++ не соответствует стандарту ISO C++ при реализации спецификаций исключений.*

| Спецификация исключений | Назначение |
|---|---|
| throw() | Функция не вызывает исключений. Однако если исключение выдается функцией, помеченной атрибутом throw(), компилятор Visual C++ не вызовет функцию unexpected (дополнительные сведения см. в разделах unexpected (CRT) и unexpected (<exception>)). Если функция помечена атрибутом throw(), компилятор Visual C++ предполагает, что она не создает исключений C++, и формирует код соответствующим образом. Из-за оптимизации кода, которую может выполнять компилятор C++ (на основе допущения, что функция не создает исключений C++), создание исключения такой функцией может привести к неправильному выполнению программы. |
| throw (...) | Функция может создавать исключения. |
| throw (тип) | Функция может создавать исключения типа type. Однако в .NET для Visual C++ это интерпретируется как throw(...). |

# Спецификации исключений

*Спецификации исключений должны подчиняться следующим правилам:*

## Исключения и сигнатуры функций

Спецификация исключений не считается частью сигнатуры функции. Другими словами, нельзя иметь две функции с совпадающим интерфейсом, отличающиеся лишь спецификацией исключений.

Две следующие функции не могут сосуществовать в программе:

```
void f1(int) throw();
void f1(int) throw(Exception);   // Повторяющаяся сигнатура!
```

# Спецификации исключений

*Спецификации исключений должны подчиняться следующим правилам:*

Спецификация исключений для виртуальных функций Вспомним об отличиях между перегрузкой (overloading) и переопределением (overriding). Если виртуальная функция в производном классе объявляется с новой сигнатурой, отсутствующей в базовом классе, эта функция скрывает все одноименные функции базового класса. Аналогичный принцип действует и для спецификаций исключений.

```
class Obj {
public:
   virtual Fn() throw(int);
};
class Bar : public Obj {
public:
   virtual Fn() throw(char*);
};
```

Компилятор косо посмотрит на это, но откомпилирует. В результате тот, кто имеет дело с Obj*, будет ожидать исключения типа int, не зная, что на самом деле он имеет дело с объектом Bar, инициирующим нечто совершенно иное.

Вывод: не следует изменять спецификацию исключений виртуальной функции в производных классах. Только так удается сохранить контракт между клиентами и базовым классом, согласно которому должны инициироваться только исключения определенного типа.

# Спецификации исключений

*Спецификации исключений должны подчиняться следующим правилам:*

Непредусмотренные исключения

Если инициированное исключение отсутствует в спецификации исключений внешней функции, программа вызывает функцию с именем **unexpected()**. По умолчанию затем вызывается функция **terminate()**, о которой будет рассказано ниже, но можно определить собственное поведение.

Соответствующие интерфейсы из заголовочного файла except выглядят так:

```
typedef void (*unexpected_function)();
unexpected_function set_unexpected(unexpected_function excpected_func);
```

В строке *typedef...* объявляется интерфейс к вашей функции. Функция **set_unexpected()** получает функцию этого типа и организует ее вызов вместо функции по умолчанию.

Функция **set_unexpected()** возвращает текущий обработчик непредусмотренных исключений. Это позволяет временно установить свой обработчик таких исключений, а потом восстановить прежний.

# Спецификации исключений

*Спецификации исключений должны подчиняться следующим правилам:*

Непредусмотренные исключения (продолжение)

```
unexpected_function my_handler(void) {
        // Обработать неожиданное исключение
}
{       // Готовимся сделать нечто и устанавливаем свой обработчик
    unexpected_function old_handler =
                                    set_unexpected(my_handler);
        // Выполняем опасный участок и возвращаем старый обработчик
    set_unexpected(old_handler);
}
```

Функция-обработчик не может нормально возвращать управление вызывающей программе, если в ней встречается оператор **return** или при выходе из области действия функции результаты будут неопределенными. Тем не менее, из функции можно запустить исключение и продолжить поиск перехватчика, подходящего для нового исключения.

# Спецификации исключений

*Спецификации исключений должны подчиняться следующим правилам:*

Непредусмотренные исключения (фактически в Microsoft)

```
#include<exception>
#include<iostream>
using namespace std;
void unfunction( )
{
    cout << "I'll be back." << endl;
    terminate( );
}
int main( )
{
    unexpected_handler oldHand = set_unexpected( unfunction );
    unexpected( );
}
```

Фактически текущая версия не реализует описанные процедуры.

Приводимый пример показывает, как можно вызвать функцию **unexpected()** напрямую из программы.

# Спецификации исключений

## Непредусмотренные исключения - завершение

Функция **unexpected( )** может быть завершена одним из трех способов:

- Генерацией (throwing) объекта типа из описанных в спецификации исключений или объекта любого типа, если обработчик непредусмотренных исключений вызывается непосредственно из программы.
- Генерацией исключения стандартного типа **bad_exception**.
- Вызовом одной из функций **terminate()**, **abort()** или **exit(int)**.

Встроенная по умолчанию программа реализации функции **unexpected( )** вызывает функцию **terminate()**.

# Если исключение не перехвачено

Если для исключения не найдется ни одного обработчика, по умолчанию вызывается глобальная функция **terminate()**.

Другими словами - если оператор **throw** использовать вне контролируемого блока, то вызывается специальная функция **terminate()**, завершающая выполнение программы.

По умолчанию **terminate()** в конечном счете вызывает библиотечную функцию **abort()**, и дело кончается аварийным завершением всей программы. Можно вмешаться и установить собственную функцию завершения с помощью библиотечной функции **set_terminate()**.

# Если исключение не перехвачено (окончание)

Функция **set_terminate()** устанавливает функцию завершения, которую вместо функции **abort()** вызывает функция **terminate()**. Функция **set_terminate()** возвращает текущую функцию завершения, которую позднее можно восстановить повторным вызовом **set_terminate()**.

В целом все повторяет ситуацию с функцией **unexpected()** .

(Кстати для Microsoft здесь все работает).

# Вложенная обработка исключений

Допускается вложение блоков **try/catch**.

```
{
 try {
  try {
   try {
     // Ненадежный
     // фрагмент
   }
   catch(...) {
   }
  }
  catch(...) {
  }
 }
 catch(...) {
 }
}
```

При вложении контролируемых блоков исключение, возникшее во внутреннем блоке, последовательно "просматривает" обработчики, переходя от внутреннего (вложенного) блока к внешнему до тех пор, пока не будет найдена подходящая процедура обработки.

Создавать подобные конструкции приходится довольно редко, но иногда возникает необходимость в разделении стековых объектов по разным областям действия.

# Вложенная обработка исключений - пример

```
#include <iostream>
using namespace std;
void compare(int k)  // Функция, генерирующая исключения.
{
        if (k%2 != 0) throw k;          // Нечетное значение.
        else  throw "Even";             // Четное значение.
}
void GG (int j)              // Функция с контролем и обработкой исключений
{
    try
    {   try { compare(j); }          // Вложенный контролируемый блок.
    catch (int n)
      { cout << "\nOdd";  throw; }     // Ретрансляция исключения.
    catch (const char  *)   { cout << "\nEven"; }
    }                                   // Конец внешнего контролируемого блока.
// Обработка ретранслированного исключения:
    catch (int i)  {   cout <<"\nResult =  " << i;   }
}   // Конец функции GG().
void main()
{   GG(4);   GG(7);  }
```

```
Even
Odd
Result = 7
```

# Внешние исключения не перехватываются!

Вы можете перехватить любое исключение, инициированное посредством **throw**. Тем не менее, существуют и другие исключения, которые не удается перехватить переносимыми способами.

Например, если пользователь применяет для завершения программы комбинацию клавиш с правым Ctrl, нет гарантии, что операционная система сгенерирует исключение, которое может быть перехвачено вашими обработчиками. Вообще говоря, обработка исключений относится только к исключениям, сгенерированным программой; все остальное непереносимо.

Механизм исключений предназначен ***только для синхронных событий***, то есть таких, которые порождаются в результате работы самой программы.

# Конструкторы и деструкторы

Одно из принципиальных достоинств стандартной схемы обработки исключений — раскрутка стека (*unwinding the stack*). При запуске исключения автоматически вызываются деструкторы всех стековых объектов между **throw** и **catch**.

```
void fn() throw(int) {
   Obj  aObj;
   // Что-то не так! Предусматриваем генерацию исключения
   throw(bad_news);
}
```

Когда возникает исключение, до передачи стека соответствующему обработчику будет вызван деструктор aObj.

# Конструкторы и деструкторы - 2

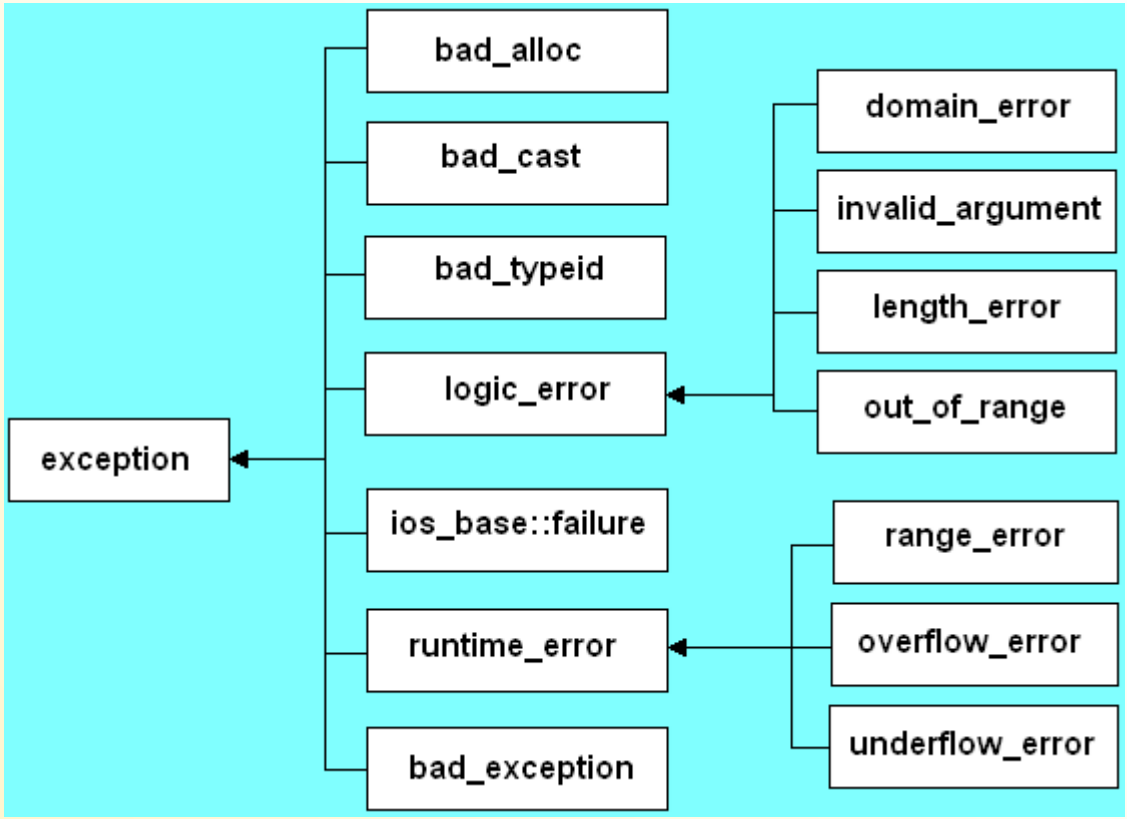
Тот же принцип действует и для try-блока вызывающей стороны.

```
{
   try {
      Bar b;
      fn(); // Вызывает исключение
   }
   catch(int exception) {
     // Перед тем, как мы попадем сюда, будет вызван деструктор b
   }
}
```

Вообще говоря, гарантируется вызов деструкторов всех стековых объектов, сконструированных с начала выполнения try-блока. Это может пригодиться для закрытия открытых файлов, предотвращения утечки памяти или для других целей. Тем не менее, дело не обходится без некоторых нюансов.

# Стандартные классы исключений

Все исключения, генерируемые языком или библиотекой, происходят от единого предка – базового класса exception. Стандартные классы исключений делятся на три категории:




- исключения языковой поддержки;
- исключения стандартной библиотеки C++;
- исключения внешних ошибок.

# Классы исключений языковой поддержки

- Исключение класса bad_alloc генерируется при неудачном выполнении глобального оператора **new** (кроме версии **new** с запретом исключений).
- Исключение класса bad_cast генерируется оператором **dynamic_cast**, если преобразование типа по ссылке во время выполнения завершается неудачей.
- Исключение класса bad_typedid генерируется оператором **typeid**, предназначенным для идентификации типов во время выполнения. Если аргументом оператора является нуль или null-указатель, то вырабатывается исключение.
- Исключение класса bad_exception предназначено для обработки непредвиденных исключений. В его обработке задействована функция **unexpected()**.

# Классы исключений стандартной библиотеки

Такие исключения обычно бывают производными от класса **logic_error**. К категории логических ошибок относятся ошибки, которые (хотя бы теоретически) можно предотвратить, например дополнительной проверкой аргументов функции. В частности, к логическим ошибкам относятся нарушение логических предусловий или инварианта класса.

- Исключение класса invalid_argument сообщает о недопустимых значениях аргументов, например, когда битовые поля инициализируются данными при помощи char со значениями, отличными от 0 и 1.
- Исключение класса length_error сообщает о попытке выполнения операции, нарушающей ограничения допустимого максимального размера, например, при присоединении к строке слишком большого количества символов.

# Классы исключений стандартной библиотеки

- Исключение класса out_of_range сообщает о том, что аргумент не входит в интервал допустимых значений, например, неправильный индекс массива.
- Исключение класса ios_base::failure определено в подсистеме организации потокового ввода/вывода и обычно генерируется при изменении состояния потока из-за ошибки или достижения конца файла.

# Классы исключений для внешних ошибок

Исключения, производные от класса **runtime_error** сообщают о событиях, не контролируемых программой.

- Исключение класса range_error сообщает об ошибках выхода за пределы допустимого интервала во внутренних вычислениях.
- Исключение класса overflow_error сообщает о математическом переполнении.
- Исключение класса underflow_error сообщает о математической потери значимости.

# Заголовочные файлы классов исключений

Базовые классы **exception** и **bad_exception** определяются в заголовочном файле <exception>.

Класс **bad_alloc** определяется в заголовочном файле <new>.

Классы **bad_cast** и **bad_typeid** определены в заголовочном файле <typeinfo>.

Класс **ios_base::failure** определяется в заголовочном файле <ios>.

Все остальные классы определены в заголовочном файле <stdexcept>.

# Члены классов исключений

Обработка исключений в секциях **catch** обычно происходит через интерфейс исключений. Интерфейс всех стандартных классов исключений состоит из единственной функции what(). Эта функция возвращает дополнительную информацию о исключении в форме текстовой строки:

```
using namespace std {
  class exception {
    virtual const char * what() const throw();
    . . .
  };
}
```

Никакой другой дополнительной информации (кроме той, что дает what()) об исключении из них достать невозможно!

Содержимое строки определяется реализацией. Остальные члены классов предназначены для создания, копирования, присваивания и уничтожения объектов исключений.

# Использование классов исключений

В пользовательских программах допустима генерация исключений класса **logic_error** (и всех производных классов), класса **runtime_error** (и всех производных классов) , а также класса **ios_base:failure**. Прочие виды исключений генерировать нельзя!

```
try
{
. . .
}
catch (const std::exception& error)
{
        // Вывод сообщения об ошибке ( что дает реализация)
  std:: cerr << error.what() << endl;
}
```

# Классы, производные от стандартных исключений

При наследовании всегда необходимо обеспечить работу функции **what()**.

```
namespace MyLib {
  // Пользовательский класс исключений
  class MyProblem : public std::exception {
    public:
      MyProblem(...) { . . . }  // Конструктор
        . . .
      virtual const char * what() const throw()    // Новая what
      { . . . }
  };
  . . .
  void func()
 {
    . . .
    throw  MyProblem(...); // Создание и генерация исключения
  }
}   // Конец пространства имен MyLib
```

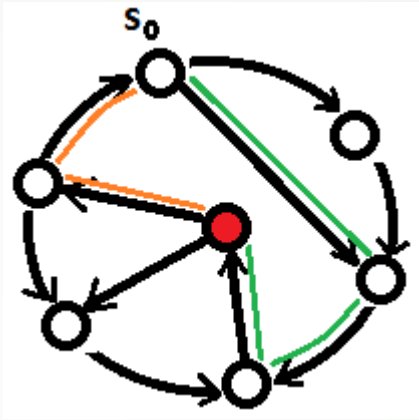
# Классы, производные от стандартных исключений

Еще один вариант – наследник от класса, конструктор которого имеет параметр - строка для функции what():

```
namespace MyLib {
  // Пользовательский класс исключений
  class MyRangeProblem : public std::out_of_range {
     public:
        MyRangeProblem(const string& S): out_of_range (S) { }
            . . .                                          // Конструктор
  };
  void func()
  {
     . . .
     throw  MyRangeProblem(" My range problem");
                                        // Создание и генерация исключения
  }
}   // Конец пространства имен MyLib
```

# Использование исключений

Предположим, требуется построить программу для нахождения совокупности путей графа, содержащих все его дуги




```
// Нахождение всех допустимых контуров
vector<int> PathSrc;
try { // Для прерывания процесса после покрытия всех дуг

   GetCycles(0, Matr, PathSrc);

}
catch(int e)   // Перехват после прерывания
{
      if (e>0)
          ArcsNotYetCovered();
}
```

```
0110000
0010000
0001000
0000100
0000011
0001000
1000010
```