# Глава 7. **Исключения**

МГТУ им. Н.Э. Баумана
Факультет Информатика и системы управления
Кафедра Компьютерные системы и сети
Лектор: д.т.н., проф.
Иванова Галина Сергеевна

# 7.1 Механизм исключений C++

В C и C++ используются разные стандарты обработки исключений.

***Генерация исключений***:

**`throw`** [<Тип>](<Аргументы>);

где <тип> – тип (чаще класс) генерируемого значения; если тип не указан, то компилятор определяет его исходя из типа аргумента (обычно это один из встроенных типов);

<аргументы> – одно или несколько выражений, значения которых будут использованы для инициализации генерируемого объекта.

**Примеры**:

```
throw ("Неверный параметр");  /* тип  const char * с указанным в
                                 кавычках значением  */
throw (221); /* тип const int с указанным  значением */

class E {   //класс исключения
    public: int num;            // номер исключения
          E(int n): num(n){}    // конструктор класса
   }
…
throw E(5);  //  класс E
```

# Перехват и обработка исключений

**try** {<Защищенный код>}
**catch** (<Ссылка на тип>){<Обработка исключений>}

При этом:

1) исключение типа T будет перехватываться обработчиками типов **`T`**, **`const T`**, **`T&`** или **`const T&`**;
2) обработчики типа базового класса перехватывают исключения типа производных классов;
3) обработчики типа **`void*`** перехватывают все исключения типа указателя.

Блок **`catch`**, для которого в качестве типа указано «…» обрабатывает исключения всех типов.

**Примеры:**

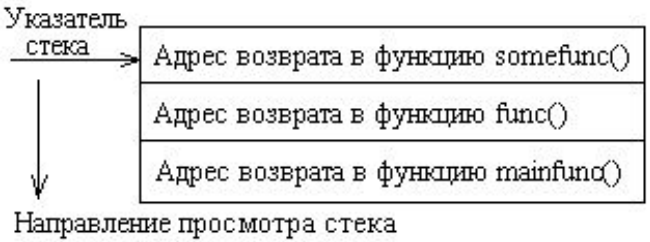
```
try {<Операторы>} // выполняемый фрагмент программы
catch (EConvert& A){<Операторы>} /* перехват исключений
                                    типа EConvert */
catch (char* Mes){<Операторы>} //перехват исключений char*
catch(…) {<Операторы>}         //перехват остальных исключений
```

# Возобновление исключения

Если перехваченное исключение не может быть обработано, то оно возобновляется.

**Пример**:

```
class E{};                    // класс исключения
void somefunc()
     { if(<условие> throw Out();   }// генерация исключения
void func()
   {   try   { somefunc(true);   }
       catch(E& e){ if (<условие>) throw; } /* если здесь
                исключение обработать нельзя, то возобновляем его  */ }
void  mainfunc()
   {   try   { func();}
       catch(E& e){ … }     } // здесь обрабатываем исключение
```



# Доступ к аргументам исключения

Использование имени переменной в качестве параметра оператора catch позволяет операторам обработки получить доступ к аргументам исключения через указанное имя.

**Пример:**

```
class E   //класс исключения
{  public: int num;              // номер исключения
          E(int n): num(n){}    // конструктор
}
   …
try { ...
throw E(5);          // генерируемое исключение
   ...}
catch (E& e){if (e.num==5) {…}} // получен доступ к полю
```

# Последовательность обработки исключения

- при генерации исключения происходит конструирование временного объекта исключения;
- выполняется поиск обработчика исключения;
- при нахождении обработчика создается копия объекта с указанным именем;
- уничтожается временный объект исключения;
- выполняется обработка исключения;
- уничтожается копия исключения с указанным именем.

Поскольку обработчику передается копия объекта исключения, желательно в классе исключения с динамическими полями предусмотреть копирующий конструктор и деструктор.

# Обработка объекта исключения (Ex7_1)

```
#include "stdafx.h"
#include <stdio.h>
class MyException
{ protected: int nError;
  public:  MyException(int nErr) {nError=nErr;}
       ~MyException(){puts("destructor");}
        void ErrorPut(){printf("Error %d.\n",nError);}
};

int main(int argc, char* argv[])
{
   try
   {
     throw MyException(5);
   }
   catch(MyException E){E.ErrorPut();}
   puts("program");
   return 0;
}
```

```
destructor
Error 5.
destructor
destructor
program
Press any key to
continue
```

# Спецификация исключений

При объявлении функции можно указать, какие исключения она может генерировать:

**throw**(<тип>,<тип>…).

Пример:

```
void func() throw(char*,int){…} /*функция может генерировать
                                 исключения char* и int   */
```

Спецификация исключений не считается частью типа функции и, следовательно, ее можно изменить при переопределении:

```
class ALPHA
{ public:  struct ALPHA_ERR{};
  virtual void vfunc() throw(ALPHA_ERR){}
};
class BETA : public ALPHA
{ public:
        void vfunc() throw(char *) {}
};
```

# Уничтожение локальных переменных при обработке исключения (Ex7_2)

```
#include "stdafx.h"
#include <iostream.h>
void MyFunc( void );
class CTest
{public: CTest(){};
         ~CTest(){};
         const char *ShowReason() const
            { return "Exception in CTest class."; }
};
class CDtorDemo
{public:  CDtorDemo();
          ~CDtorDemo();
};
CDtorDemo::CDtorDemo()
{    cout << "Constructing CDtorDemo." << endl;}
CDtorDemo::~CDtorDemo()
{    cout << "Destructing CDtorDemo." << endl;}
```

При «раскручивании» стека локальные переменные уничтожаются

# Уничтожение локальных переменных при обработке исключения (2)

```
void MyFunc()
{CDtorDemo D;
    cout<<"In MyFunc().Throwing CTest exception."<<endl;
    throw CTest();
}
int main()
{   cout << "In main." << endl;
    try
      { cout<<"In try block, calling MyFunc()."<<endl;
        MyFunc(); }
      catch(CTest E)
      {   cout << "In catch handler." << endl;
          cout << "Caught CTest exception type: ";
          cout << E.ShowReason() << endl;
      }
      catch( char *str )
        {cout<<"Caught other exceptions: "<<str<<endl;}
    cout<<"Back in main. Execution resumes here."<<endl;
    return 0;}
```

# Результат

```
In main.
In try block, calling MyFunc().
Constructing CDtorDemo.
In MyFunc(). Throwing CTest exception.
Destructing CDtorDemo.
In catch handler.
Caught CTest exception type: Exception in CTest class.
Back in main. Execution resumes here.
Press any key to continue
```

# Обработка непредусмотренных исключений

Для определения функции обработки непредусмороенных исключений в программе используется функция **set_unexpected**:

```
void my_unexpected()   {<обработка исключений> }
…
set_unexpected(my_unexpected);
```

Функция set_unexpected() возвращает старый адрес функции – обработчика непредусмотренных исключений.

Если обработчик непредусмотренных исключений отсутствует, то вызывается функция **terminate**(). По умолчанию эта функция вызывает функцию **abort**(), которая аварийно завершает текущий процесс.

Для определения собственной функции завершения используется функция **set_terminate**():

```
void my_terminate(){<обработка завершения>}
…
set_terminate(my_terminate);
```

Функция **set_terminate()** также возвращает адрес предыдущей программы обработки завершения.

# Завершающая обработка (Ex7_3)

```
#include "stdafx.h"
#include <eh.h>       // For function prototypes
#include <iostream.h>
#include <process.h>
void term_func()
{ cout << "term_func was called by terminate." << endl;
  exit( -1 );
 }
int main()
{ try
    { set_terminate( term_func );
      throw "Out of memory!";   }
    catch( int )
    {cout << "Integer exception raised." << endl;     }
  return 0;
}
```

# 7.2 Механизм структурного управления исключениями С

Для перехвата исключения в языке С используется конструкция:

**_ _try** {<защищенный код>}

**_ _except**(<фильтрующее выражение>)

{<обработка исключений>}

Фильтрующее выражение может принимать следующие значения:

- 1 = EXCEPTION_EXECUTE_HANDLER – управление должно быть пере-дано на следующий за ним обработчик исключения (при этом по умолчанию при обратном просмотре стека вызовов активизируются деструкторы всех локальных объектов, созданных между местом генерации исключения и найденным обработчиком);
- 0 = EXCEPTION_CONTINUE_SEARCH – производится поиск другого обработчика;
- -1 = EXCEPTION_CONTINUE_EXECUTION – управление возвращается в то место, где было обнаружено исключение без обработки исключе-ния (отмена исключения).

В качестве фильтрующего выражения обычно используется функция, которая возвращает одно из указанных выше трех значений.

# Получение информации об исключении

Для получения информации об исключении используют:

**_exception_code** – возвращает код исключения.

**_exception_info** – возвращает указатель на структуру **EXCEPTION_POINTERS**, содержащую описание исключения:

```
struct exception_pointers {
      EXCEPTION_RECORD *ExceptionRecord,
      CONTEXT *ContextRecord }
struct EXCEPTION_RECORD
 {     DWORD ExceptionCode;     // код завершения
       DWORD ExceptionFlags;    // флаг возобновления
       struct EXCEPTION_RECORD *ExceptionRecord;
       void *ExceptionAddress;     // адрес исключения
       DWORD NumberParameters;  // количество аргументов
       DWORD ExceptionInformation
       [EXCEPTION_MAXIMUM_PARAMETERS];  /* адрес массива
                                           параметров */
 };
```

# Получение информации об исключении (2)

Существует ограничение на вызов этих функций: они могут вызываться только непосредственно из блока _ _except(). Фильтрующая функция не может вызывать **_exception_info**, но результат этого вызова можно передать в качестве параметра. Так например:

```
__except (filter_func(xp = _exception_code))
  {/* получение информации об исключении */ }
```

или с использованием составного оператора:

```
__except((xp = _exception_info),
                              filter_func(xp))
```

.

# Обработка аппаратных и программных исключений Windows (Ex7_4)

```
#include "stdafx.h"
#include <EXCPT.H>
#include <stdio.h>


void main()
{
    int* p = 0x00000000;    // NULL
    puts("begin");
    __try{
        puts("in try");
        *p = 13;     // генерация исключения
            }
    __except(printf("code = %x\n",_exception_code()),1)
                                                    {  }
    puts("end");
}
```

```
begin
in try
code = c0000005
end
Press any key to
continue
```

# Генерация исключений в С

Для генерации исключения используется функция

**void RaiseException(DWORD <**код исключения**>,**
**DWORD <**флаг**>,**
**DWORD <**количество аргументов**>,**
**const DWORD *<**адрес массива 32 разрядных аргументов**>);**

где <код исключения> – число следующего вида:

биты 30-31: 11 – ошибка; 01 – информация; 10 – предупреждение;
бит 29: 1 – не системный код;
бит 28: 0 – резерв;

<флаг> может принимать значения:

**EXCEPTION_CONTINUABLE (0)** – обработка возобновима;
**EXCEPTION_NONCONTINUABLE** – обработка не возобновима (любая попытка продолжить процесс вызовет соответствующее прерывание).

**Пример:**

```
#define STATUS_1  0xE0000001
   ...
RaiseException(STATUS_1, 0, 0, 0);
```

# Завершающая обработка

Структурное управление исключениями поддерживает также завершающую конструкцию, которая выполняется независимо от того, было ли обнаружено исключение при выполнении защищенного блока:

_ _**try** {<защищенный блок> }

_ _**finally** {<завершающая обработка>}

Пример:

# Совместное использование обычной и завершающей обработки исключений (Ex7_5)

```
#include "stdafx.h"
#include "stdio.h"
void main()
{   int* p = 0x00000000;// NULL
    puts("Begin");
    __try{
            puts("in try1");
            __try{
                   puts("in try2");
                   *p = 13;    // генерация исключения
                  }
            __finally{ puts("in finally"); }
          }
    __except(puts("in filter"),1){puts("in except");}
    puts(«end");
}
```

```
hello
in try1
in try2
in filter
in finally
in except
world
Press any key to
continue
```

# 7.3 Совместное использование различных механизмов обработки исключений

1) исключения Win32 можно обрабатывать только try…_ _except (C++) или _ _try…_ _except (C) или соответственно try…_ _finally (C++) или _ _try…_ _finally (C); оператор **`catch`** эти исключения игнорирует;

2) неперехваченные исключения Win32 не обрабатываются функцией обработки неперехваченных исключений и функцией **`terminate()`**, а передаются операционной системе, что обычно приводит к аварийному завершению приложения;

3) обработчики структурных исключений не получают копии объекта исключения, так как он не создается, а для получения информации об исключении используют специальные функции **`_exception_code`** и **`_exception_info`**.

Если возникает необходимость перехвата структурных исключений и исключений C++ для одной последовательности операторов, то соответствующие конструкции вкладываются одна в другую.

# Пример совместного использования механизмов исключения (Ex7_6)

```
#include "stdafx.h"
#include <string.h>
#include <stdio.h>
class MyException          // класс исключения
 { private:  char* what; // динамическое поле сообщения
    public:  MyException(char* s);
             MyException(const MyException& e );
             ~MyException();
             char* msg()const;
 };
MyException::MyException(char* s = "Unknown")
               { what = strdup(s);  }
MyException::MyException(const MyException& e )
               { what = strdup(e.what);}
MyException::~MyException()
               { delete[] what;  }
char* MyException::msg() const
               { return what; }
```

# Пример совместного использования механизмов исключения (2)

В пределах функции нельзя использовать исключения разного типа!

```
void f()
{int *p=NULL;
  __try    { *p=3;}
  __except(1){throw(MyException("Wrong pointer"));}}

void f1()
   {try { f();}
    catch(const MyException& e){ puts(e.msg());}
   }

int main(int argc, char* argv[])
 {__try
    {f1();}
  __finally  { puts("end"); }
  return 0;
 }
```

```
Wrong pointer
end
Press any key to
continue
```

# Обработка структурных исключений как исключений C++

Для преобразования структурных исключений в исключения C++ можно использовать функцию **_set_se_translator()**:

```
typedef void (* translator_function)
            ( unsigned int, struct _EXCEPTION_POINTERS* );
translator_function _set_se_translator
                    (translator_function se_trans_func);
```

В качестве параметра функции **_set_se_translator** необходимо передать адрес функции-переходника, которая назначает для структурных исключений вызов соответствующих исключений C++, например:

```
void trans_func( unsigned int u,EXCEPTION_POINTERS* pExp)
    {throw SE_Exception(u);}
```

Функция возвращает указатель на предыдущую функцию **_set_se_translator()**

# Трансляция структурных исключений (Ex7_7)

```
#include "stdafx.h"
#include <stdio.h>
#include <windows.h>
#include <eh.h>

class SE_Exception  // класс для обработки структурных
                    // исключений
{private:  unsigned int nSE;
 public:
    SE_Exception() {}
    SE_Exception( unsigned int n ) : nSE( n ) {}
    ~SE_Exception() {}
    unsigned int getSeNumber() { return nSE; }
};

void SEFunc() // пример функции с исключением
{   _ _try {  int x, y=0; x = 5 / y; } // исключение!
    _ _finally  { printf( "In finally\n" ); }
}
```

# Трансляция структурных исключений (2)

```
void trans_func( unsigned int u, EXCEPTION_POINTERS* pExp)
{   printf( "In trans_func.\n" );
    throw SE_Exception(u);
}
void main( void)
{ try
    {   _set_se_translator( trans_func );
        SEFunc();
    }
    catch( SE_Exception e )
    {   printf( "%x.\n",e.getSeNumber() );   }
}
```

```
In trans_func.
In finally
c0000094.
Press any key to continue
```