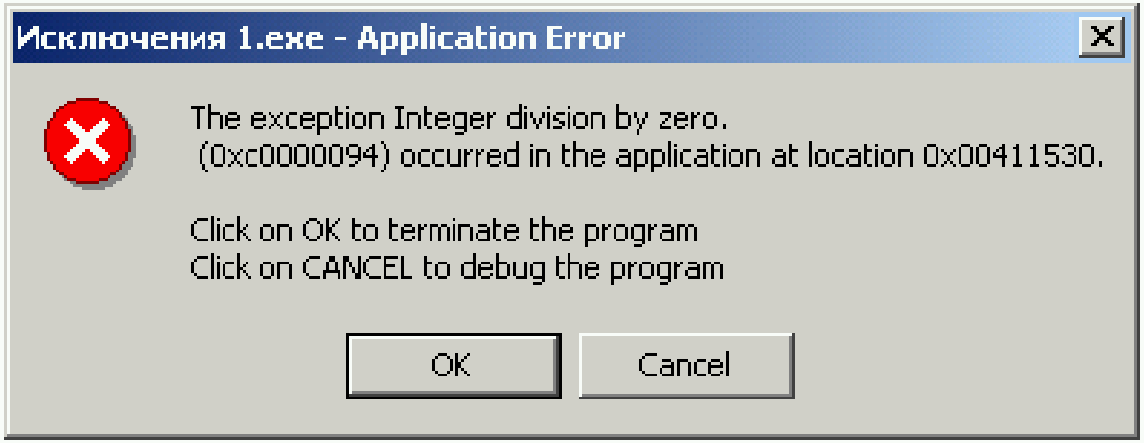
ЛЕКЦИЯ 20
8 ноября 2016 года

Обработка исключительных ситуаций Часть 1

- При работе программ возникают т.н. *исключительные ситуации*, когда дальнейшее нормальное выполнение приложения становится невозможным. Причиной исключительных ситуаций могут быть как ошибки в программе, так и неправильные действия пользователя, неверные данные и т.д. Программист должен иметь в своем распоряжении средства для обнаружения и обработки таких ситуаций.

Традиционные средства обработки, связанные с многочисленными проверками, делают любую мало-мальски серьезную программу нечитабельной. Кроме того, предусмотреть обработку абсолютно всех таких ситуаций практически невозможно. Так, выполняя следующий фрагмент программы на экране получим следующие картинки:

```
#include <iostream>
using namespace std;
int main()
{
     int x=100;
     int y=0;
     cout << "x/2=" << x/2 << endl;
     cout << "x/y=" ;
     cout << x/y << endl;
     cout << "STOP" << endl;
     return 0;
}
```



Не исключено, что на некоторых компьютерах первую картинку не успеете заметить, а программа зависнет.
Ошибка здесь – в делении на нуль.
(К критическим ситуациям могут приводить обращение по несуществующему адресу памяти, запрос на выделение памяти больше, чем имеется и т.д.)

Обычно ошибки приводят к завершению программы с системным сообщением об ошибке.

Исключения позволяют логически разделить вычислительный процесс на две части:

- обнаружение аварийной ситуации;
- ее обработка.

Простейшая обработка исключений

Язык C++ включает следующие возможности для работы с исключениями:

- создание защищенных блоков (**try**-блок) и перехват исключений (**catch**-блок);
- инициализация исключений (инструкция **throw**).

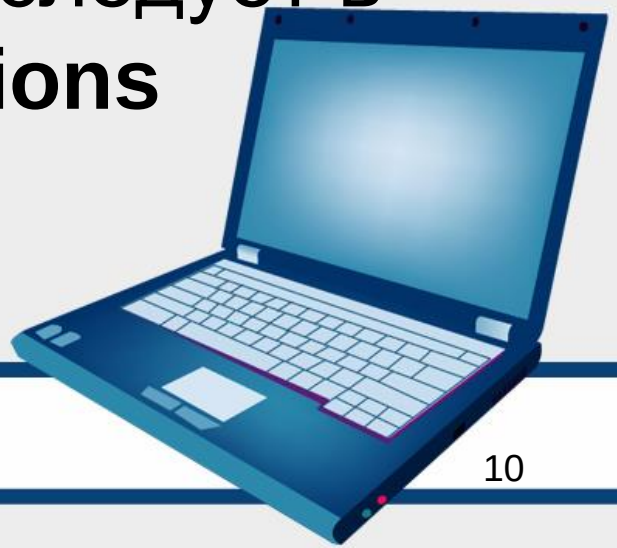
В системе программирования Visual Studio различают два типа исключений:

- ***исключения C++;***
- ***системные исключения***.

Первый тип исключений генерируется в самой программе инструкцией **throw**. Второй тип исключений генерируется операционной системой. Такие исключения также называют ***асинхронными*** (asynchronous exceptions).

Для обеспечения перехвата исключений C++, необходимо включить в **Enable C++ Exceptions** режим компиляции /**EHsc** , а для перехвата исключений любого типа – режим /**EHa**.

Кроме того, для перехвата системных исключений, связанных с обработкой данных с плавающей точкой, следует в **Enable Floating Point Exceptions** включить режим /**fp:except**.

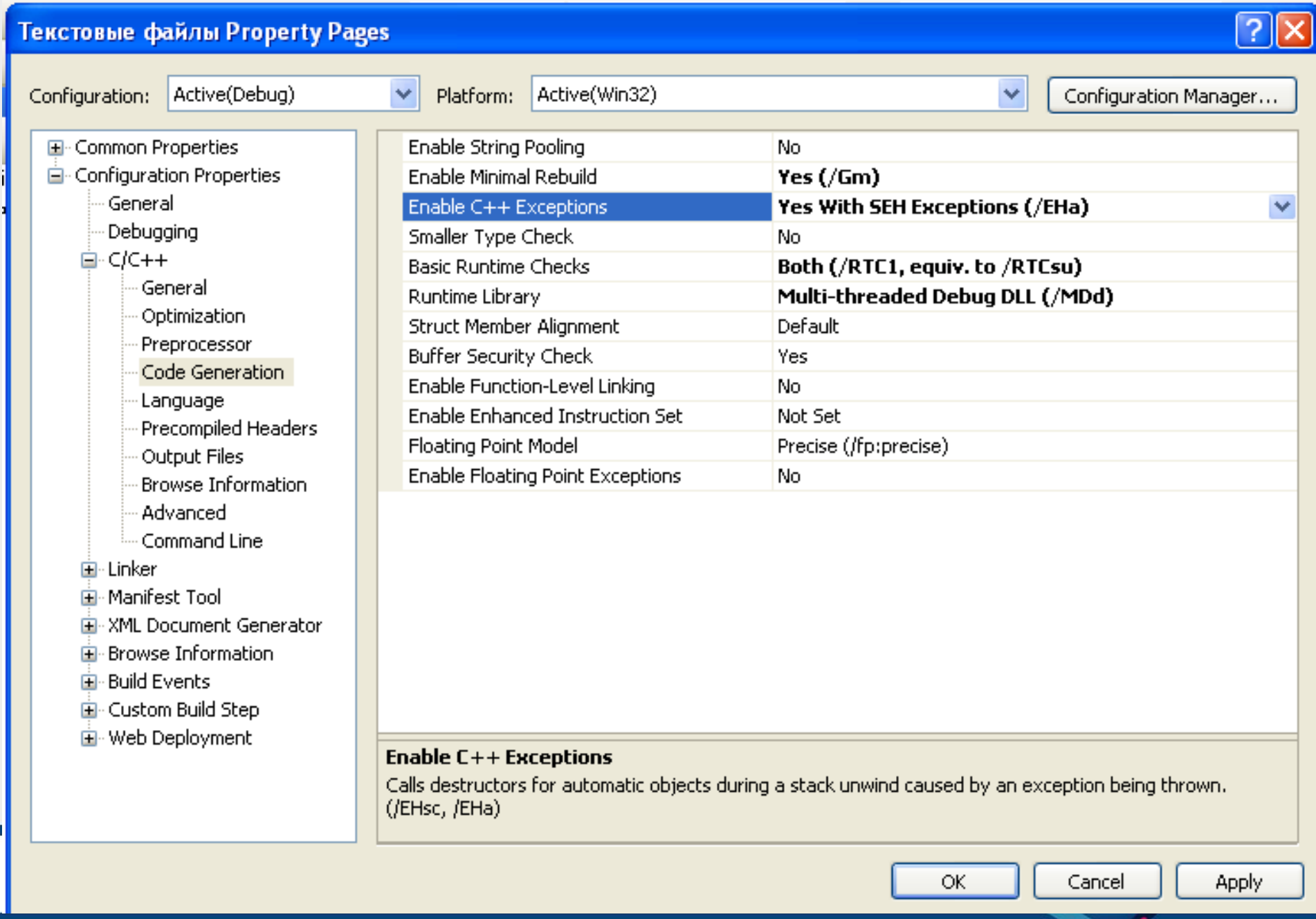


Переключение этих режимов выполняется в режиме изменения свойств проекта:

1. Open the project's **Property Pages** dialog box.
2. Click the **C/C++** folder.
3. Click the **Code Generation** property page.
4. Modify the **Enable C++ Exceptions** property.

Или это можно сделать по-другому:

1. Click the **C/C++** folder.
2. Click the **Code Generation** property page.
3. Set **Enable C++ Exceptions** to **No**.
4. Click the **Command Line** property page.
5. Type the compiler option in the **Additional Options** box.



Простейший формат защищенного блока имеет вид

```
try
{операторы_защищенного_блока}
catch(...)
{обработчик_ошибочной_ситуации}
```

Многоточие является частью синтаксиса языка!

- Работа инструкции **try … catch** выполняется следующим образом. Выполняются инструкции, входящие в состав блока **try** (защищенный блок). Если при их выполнении исключение не возбуждается (в C++ чаще используется термин «выброс исключения»), то блок **catch** пропускается. При выбросе исключения выполнение защищенного блока прекращается, и начинают работать инструкции, записанные в блоке **catch.**

- Основной смысл этих инструкций – корректная обработка исключительной ситуации. Кроме того, в блок **catch** имеет смысл поместить код, который освобождает ресурсы, захваченные выполнившимися инструкциями из блока **try**.
- После окончания работы блока **catch** исключение считается обработанным, и управление передается на первую инструкцию, следующую за конструкцией **try** …**catch**.

Рассмотрим пример перехвата системного исключения «деление на ноль».

Простейший вид инструкции try …catch (системные исключения)

```
int x = 0;
try {
 cout << 2/x;  // Здесь произойдет выброс
  ……           // исключения
               // Последующие операторы
               // выполняться не будут
}
catch (...) {
  cout << "Division by zero" << endl;
}
```

Инициализация исключений и их обработка

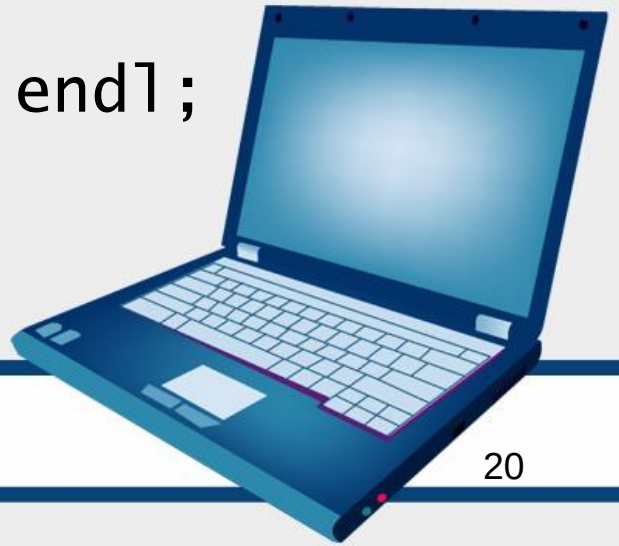
Гораздо более интересным является механизм создания собственных исключений.

Для их возбуждения используется оператор

throw *выражение*

Простейший вид инструкции try …catch (исключения языка)

```
try {
…
 throw 0;
  // Здесь произойдет выброс исключения
  // Последующие операторы
  // выполняться не будут
}
catch (...) {
 cout << “Everything fail!” << endl;
}
```



Тип выражения, указанного в операторе **throw**, определяет тип исключительной ситуации, а значение может быть передано обработчику прерываний. Этот механизм, заявленный как стандартный, представляется весьма экзотическим без использования механизма классов.

И только использование стандартных классов-исключений или разработка собственных классов позволяют в полной мере оценить все возможности такого подхода.

полный формат защищенного блока имеет вид

```
try
  {операторы_защищенного_блока}
{catch-блоки}…
```

Catch-блок имеет один из следующих форматов:

```
catch (тип)
  {обработчик_ошибочной_ситуации}
catch (тип идентификатор)
  {обработчик_ошибочной_ситуации}
catch (...)
  {обработчик_ошибочной_ситуации}
```

- Первый формат используется, если надо указать тип перехватываемого исключения, но не нужно обрабатывать связанное с этим исключением значение.
- Второй формат используется, если имя параметра (идентификатор) используется в теле обработчика для выполнения каких-либо действий, например, вывода информации об исключении.
- Третий формат оператора **`catch`** позволяет обработать все исключения (в том числе и ошибки выполнения, как в последнем примере).

Пример записи блоков catch:

```
catch (int i)
{…}
catch (const char*)
{…}
catch (OverFlow)
{…}
catch (…)
{…}
```

Обработка исключений, возбужденных оператором **throw**, идет по следующей схеме:

1. Создается статическая переменная со значением, заданным в операторе **throw.** Она будет существовать до тех пор, пока исключение не будет обработано. Если переменная-исключение является объектом класса, при ее создании работает конструктор копирования.

2. Завершается выполнение защищенного **try**-блока: раскручивается стек подпрограмм, вызываются деструкторы для тех объектов, время жизни которых истекает и т.д.

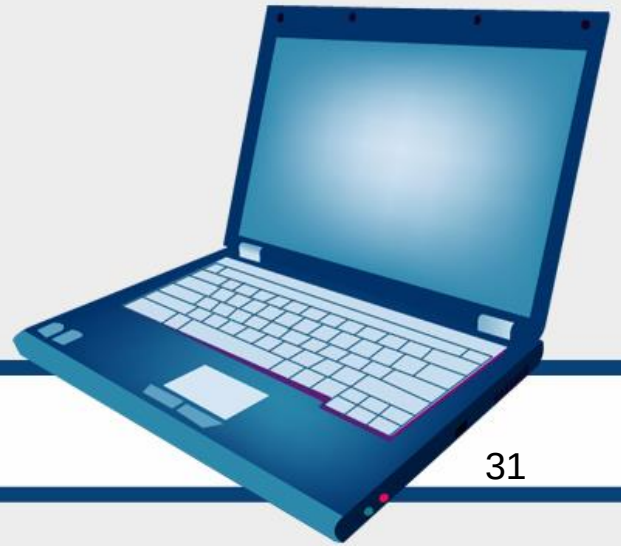
3. Выполняется поиск первого из **catch**-блоков, который пригоден для обработки созданного исключения. Поиск ведется по следующим критериям:

- если тип, указанный в **catch**-блоке, совпадает с типом созданного исключения, или является ссылкой на этот тип (параметр может быть записан в виде **T, const T, T&, const T&,** где **T** – тип исключения);
- класс, заданный в **catch**-блоке, является предком класса, заданного в **throw**, и наследование выполнялось с ключом доступа **public**;

- указатель, заданный в операторе **throw**, может быть преобразован по стандартным правилам к указателю, заданному в **catch**-блоке.
- в операторе **throw** задано многоточие.

- Если нужный обработчик найден, то ему передается управление и, при необходимости, значение оператора **throw**. Оставшиеся **catch**-блоки, относящиеся к защищенному блоку, в котором было создано исключение, игнорируются.

- Из указанных правил поиска следует, что очень важен порядок расположения **catch**-блоков. Так, блок **catch(…)** должен стоять последним в списке, а блок **catch (void *)** – после всех блоков с указательными типами.



- Если ни один из **catch**-блоков, указанных после защищенного блока, не сработал, то исключение считается необработанным. Его обработка может быть продолжена во внешних блоках **try** (если они, конечно, есть!).

- В конце оператора **catch** может стоять оператор **throw** без параметров. В этом случае работа **catch**-блока считается незавершенной а исключение – не обработанным до конца, и происходит поиск соответствующего обработчика на более высоких уровнях.

Инструкция try …catch с проверкой типа

```
try {
… // здесь исключений нет
 throw 0;
//Здесь произойдет выброс исключения!
// Последующие операторы выполняться не будут
}
catch (int) {
 cout << “Int type exception was thrown!”
      <<  endl;
}
catch (...) {
 cout << “Everything fail!” << endl;
// этот блок не будет работать
}
```

Инструкция try …catch с проверкой типа и значения

```
try {
…
 throw 0;
}
catch (int e) {
 cout <<
   “Int type exception was thrown, code is ”
   << e << endl;
}
catch (...) {
 cout << “Everything fail!” << endl;
}
```

- Если оператор **throw** был вызван вне защищенного блока (что чаще всего случается, когда исключение возбуждается в вызванной функции), или если не был найден ни один подходящий обработчик этого исключения, то вызывается стандартная функция **std::terminate()**. Она, в свою очередь, вызывает функцию **abort()** для завершения работы с приложением (то же, что и без обработки исключений).

```
int main()
{
	throw 500;
	return 0;
}
```

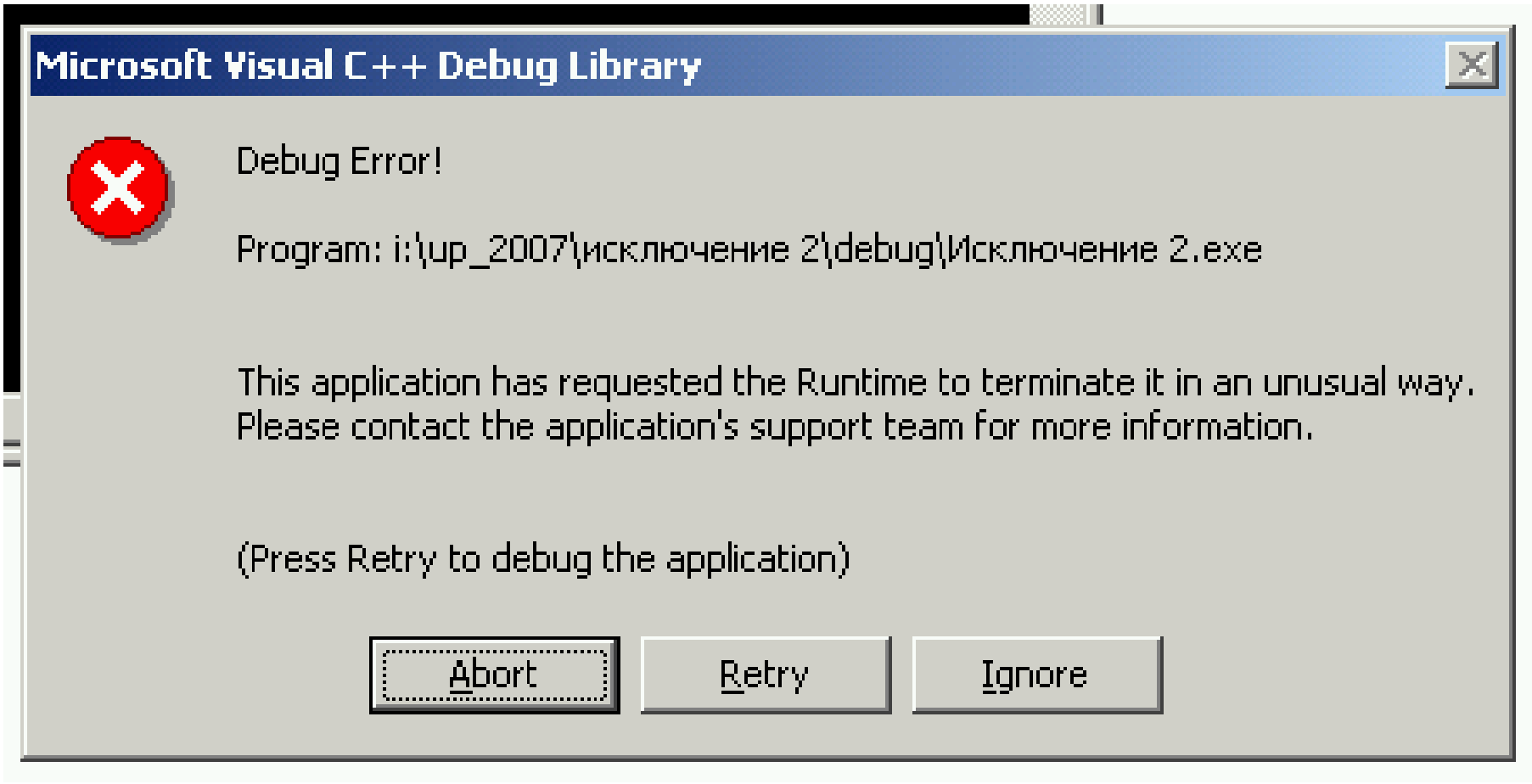


РИС.1.

- Можно заменить вызов функции abort() вызовом своего обработчика. В этом случае нужно зарегистрировать с помощью функции **set_terminate** свою функцию, которая будет выполняться перед аварийным завершением работы:

```
void MyAbort()
{
        cerr << "*** Program is terminate ***\n";
        exit(1);
}
int main()
{
        set_terminate(MyAbort);
        throw 500;
        return 0;
}
```

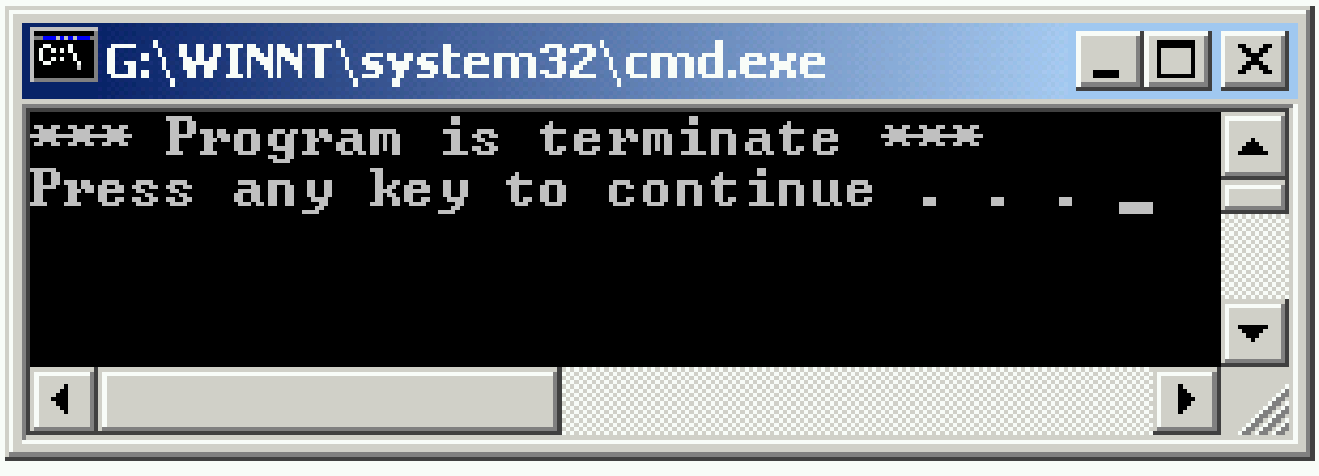


РИС.2.

Примеры обработки исключительных ситуаций

Выделенные цветом блоки не работают.

Работающие обработчики прерываний (пример 1)

```
try { …
  try {  …  throw ”Error!”;  …
  } //внутренний try
  catch (int) {… }
  catch (float) {… }…
 } //внешний try
 catch (char * c) { … }
 catch (...) { …}
```

Работающие обработчики прерываний (пример 2)

```
try { …
  try {  …
    throw ”Error!”;
    …
  } //внутренний try
  catch (char *) {… }
  catch (float) {… }
  …
} //внешний try
catch (char * c) {…}
catch (...) { …}
```

Работающие обработчики прерываний (пример 2)

```
try { …
  try {  …
    throw ”Error!”;
    …
  } //внутренний try
  catch (char *) {… }
  catch (float) {… }
  …
} //внешний try
catch (char * c) {…}
catch (...) { …}
```

Работающие обработчики прерываний (пример 3)

```
try { …
 try { …
  throw ”Error!”;   …
  } //внутренний try
  catch (char *)
  {…
    throw; // переход на более высокий уровень
  }
  catch (float) {… }
  …
 } //внешний try
 catch (char * c) {…}
 catch (...) { …}
```

Работающие обработчики прерываний (пример 4)

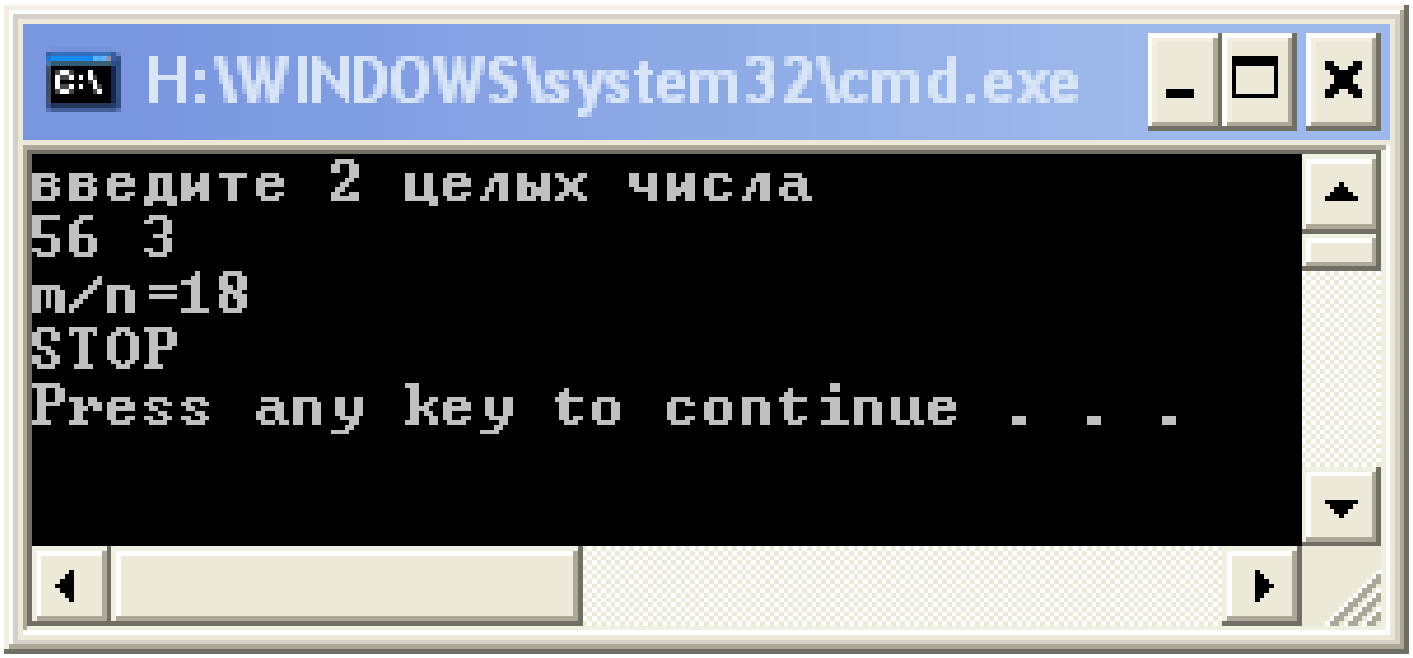
```
try{ …
  try {…
    throw ”Error!”;   …
  } //внутренний try
  catch (void *)
  {…
     throw; // переход на более высокий уровень
  }
  catch (float) {… }
  …
} //внешний try
catch (char * c) {…}
catch (...) { …}
```

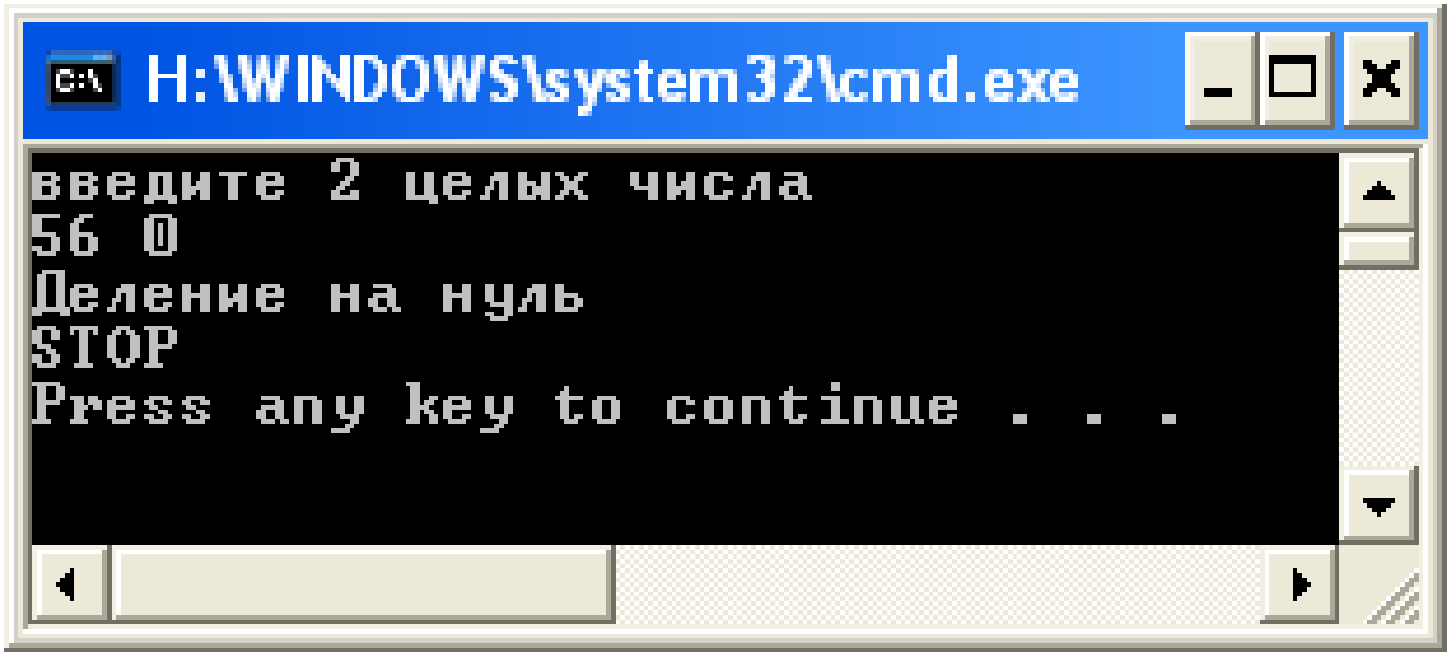
Пример 20.1.

(тип выбрасываемого исключения - строка)

```
int divide(int a, int b)
{
    if (!b)
        throw "Деление на нуль";
    return a/b;
}
```

```
int main()
{    int n, m;
     try  {
      cout<<"введите 2 целых числа»<<endl;
      cin >> m >> n;
      cout << "m/n=" <<
      divide(m,n) << endl;
     }
     catch(char* str){
          cout << str << endl;
     }
     cout << "STOP" << endl;
     return 0;
}
```


H:\WINDOWS\system32\cmd.exe
введите 2 целых числа
56 3
m/n=18
STOP
Press any key to continue . . .


H:\WINDOWS\system32\cmd.exe
введите 2 целых числа
56 0
Деление на нуль
STOP
Press any key to continue . . .

Пример 20.2. *(используется пустой тип)*

```
struct ZeroDivide {};
int divide(int a, int b)
{ if (!b) throw ZeroDivide();
   return a/b;
}
int main()
{ int n, m;
  try{
   cout<<"введите 2 целых числа "<<endl;
   cin >> m >> n;
   cout << "m/n=" << divide(m,n) << endl;
  }
  catch(ZeroDivide){
        cout << "деление на нуль"  << endl;}
  cout << "STOP" << endl;
  return 0;}
```

Пример 20.3. *(вложенные исключения)*

```
struct UnderZero{};
struct Zero{};
struct AboveZero{};
int main()
{
 setlocale(LC_ALL,".1251");
 int n, m;
 cout<<"введите целое число"<<endl;
 cin >> n;
```

```
try
{
    if (n<0) throw UnderZero();
    try
    {
     if (n>0)throw AboveZero();
     throw Zero();
    } //внутренний try
    catch(AboveZero)
    {
     cout << "число положительное" << endl;
    }
} //внешний try
```

```
  catch(UnderZero)
  {
     cout<< "число отрицательное"<< endl;
  }
  catch(Zero)
  {
     cout << "число равно 0"<< endl;
  }
  cout << "STOP" << endl;
  return 0;
}
```

Пример 20.4. *(Передача управления по иерархии обработчиков)*

```
struct FreeMemory{};
int main()
{  int *n, *m;
…
   try
   {    n=new int;
        cout << "введите n" << endl;  cin >> *n;
        try
        {       m = new int;
                cout << "введите m" << endl; cin >> *m;…
                throw FreeMemory();      //нужно освоб память
        }
        catch(FreeMemory)
        {       cout<<"m=" << *m << endl;      …
                delete m;
                throw; // нужно еще раз освоб память
        }
   }
   catch(FreeMemory)
   {    cout << "n=" << *n << endl; delete n;}
        cout << "STOP" << endl;
        return 0; }
```

КОНЕЦ ЛЕКЦИИ