Тема 10

Исключения

Исключительные ситуации

При работе программ возникают т.н. **исключительные ситуации**, когда дальнейшее нормальное выполнение приложения становится невозможным.

Причиной исключительных ситуаций могут быть:

- ошибки в программе;
- неправильные действия пользователя;
- неверные данные и т.д.

Программист должен иметь в своем распоряжении средства для обнаружения и обработки таких ситуаций.

Классификация исключительных ситуаций

В системе программирования Visual Studio 2008 (2010, 2013) различают два типа исключений:

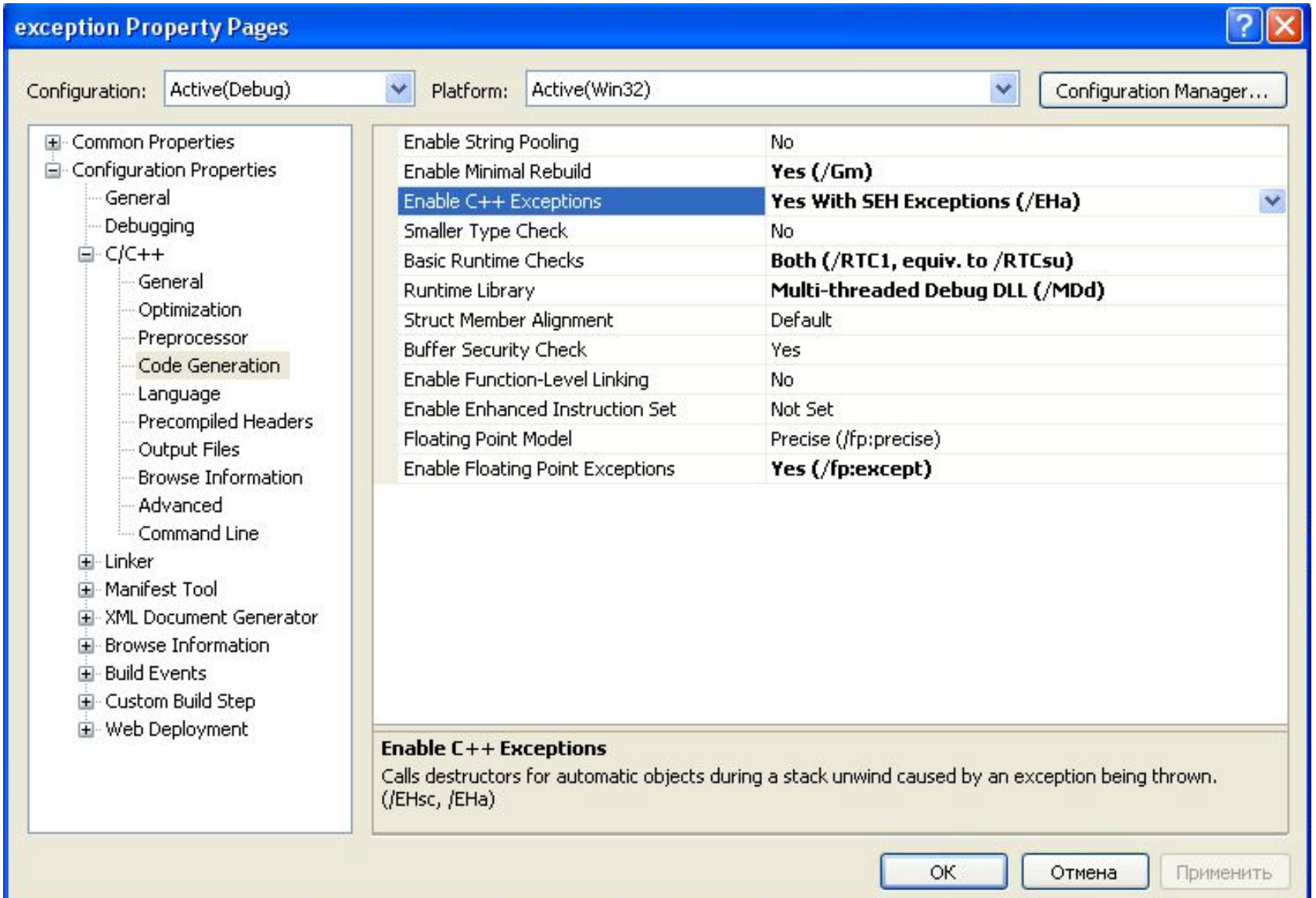
- исключения C++;
- системные исключения.

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

Режимы компиляции для работы с исключительными ситуациями

Для того чтобы обеспечить перехват исключений C++, необходимо включить 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 включить режим /fp:except.

Переключение режимов компиляции для перехвата исключений



Инструкции C++ для работы с исключительными ситуациями

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

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

Защищённый блок

Простейший формат защищенного блока имеет вид

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

Многоточие является частью синтаксиса языка!

Механизм работы защищённого блока

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.**

После окончания работы блока catch исключение считается обработанным, и управление передается на первую инструкцию, следующую за конструкцией **try** …**catch**.

Пример выброса исключения (системные исключения)

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

Для корректной работы этого примера необходимо включить режим компиляции /EHa

Возбуждение собственных исключений

Для возбуждения собственных исключений используется оператор

```
throw выражение
```

Тип выражения, указанного в операторе **throw**, определяет тип исключительной ситуации, а значение может быть передано обработчику исключений.

Полный формат защищённого блока

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

Catch-блок имеет один из следующих форматов:

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

Первый формат используется, если нам надо указать тип перехватываемого исключения, но не нужно обрабатывать связанное с этим исключением значение (это достигается при использовании второго формата оператора **catch**). Наконец, третий формат оператора **catch** позволяет обработать все исключения (в том числе и системные).

Пример выброса исключения (собственные исключения)

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

Пример выброса исключения (собственные исключения)

```
try {
…
 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
}
```

Пример выброса исключения (собственные исключения)

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

Последовательность действий при обработке исключений

- Создается статическая переменная со значением, заданным в операторе **throw.** Она будет существовать до тех пор, пока исключение не будет обработано.
- Завершается выполнение защищенного **try**-блока: раскручивается стек подпрограмм, корректно уничтожаются объекты, время жизни которых истекает и т.д.
- Выполняется поиск первого из **catch**-блоков, который пригоден для обработки созданного исключения.

Последовательность действий при обработке исключений (продолжение)

Поиск **catch**-блоков ведется по следующим критериям:

- тип, указанный в **catch**-блоке, совпадает с типом созданного исключения, или является ссылкой на этот тип;
- 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, вычисленное в операторе **throw**. Оставшиеся **catch**-блоки игнорируются.

Последовательность действий при обработке исключений (продолжение)

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.

Работающие обработчики исключений (пример 1)

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

Работающие обработчики исключений (пример 2)

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

Работающие обработчики исключений (пример 3)

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

Работающие обработчики исключений (пример 4)

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

Работающие обработчики исключений (пример 5)

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

Необработанное исключение

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 terminate(). Она, в свою очередь, вызывает функцию abort() для завершения работы с приложением.

Собственная функция аварийного завершения

Можно зарегистрировать с помощью функции set_terminate свою функцию, которая будет выполняться перед аварийным завершением работы:

```
void MyTerminate() {
   cout << "An error occured!" << endl;
   exit(-1);
}

int main ()
{
    set_terminate(MyTerminate);
…
    throw 0;
}
```

Тонкая обработка системных исключений

Режим компиляции /EHa позволяет перехватывать и обрабатывать системные исключения, возникающие в процессе работы программы. Однако обработчик таких исключений помещается в блок **catch (…)** и не дает возможности определить, какое именно исключение возникло.

Для более детальной обработки системных исключений можно воспользоваться механизмом **трансляции исключений**

Трансляция исключений

Транслятор исключений – пользовательская callback-функция, прототип которой имеет вид

```
void MyTranslator(unsigned err_code,
  _EXCEPTION_POINTERS *p);
```

Параметр err_code обозначает тип исключительной ситуации (константа EXCEPTION_INT_DIVIDE_BY_ZERO, например, обозначает попытку деления на ноль в целочисленной арифметике, а константа EXCEPTION_ACCESS_VIOLATION – попытку обратиться к запрещенному адресу памяти).

Указатель p содержит адрес структуры, содержащей дополнительную информацию об исключении.

Трансляция исключений (продолжение)

Написанный транслятор необходимо зарегистрировать вызовом функции

```
_set_se_translator(MyTranslator);
```

После этого транслятор получает управление при каждом выбросе системного исключения.

Транслятор – не обработчик исключения!

По завершению его работы выполняется стандартные действия по обработке исключения!

Преобразование системных исключений в пользовательские

```
void MyTranslator(unsigned err_code,
   _EXCEPTION_POINTERS *p) {
    throw err_code;
}
```

Теперь в блоке catch(unsigned) можно выполнить более тонкую обработку системных исключений:

```
int main() {

   int x = 0;
   int *px = NULL;

   _set_se_translator(MyTranslator);

   try {
// cout << 2/x;
    *px=0;
   }
```

Преобразование системных исключений в пользовательские (продолжение)

```
   catch (unsigned e) {
   switch (e) {
    case EXCEPTION_INT_DIVIDE_BY_ZERO:
    cout << "Division by zero" << endl;
    break;
    case EXCEPTION_ACCESS_VIOLATION:
    cout << "Invalid pointer assignement" << endl;
        break;
    }
   }
   return 0;
}
```