Санкт-Петербургский политехнический университет Петра Великого

Институт компьютерных наук и технологий

Кафедра компьютерных систем и программных технологий

**Лабораторная работа №1 (Обработка исключений в Linux)**

**Дисциплина**: Системное программирование

Выполнил студент гр. 13541/3 Смирнов М.И.

(подпись)

Руководитель Душутина Е.В.

(подпись)

“ ” 2017 г.

Санкт – Петербург

2017

**Содержание**

[1. Задание 3](#_Toc499823780)

[2. Решение 4](#_Toc499823781)

[1. Сгенерировать и обработать исключение с помощь WinAPI функций 4](#_Toc499823782)

[2. Получить код исключения с помощью GetExceptionCode двумя способами 6](#_Toc499823783)

[3. Создать собственную функцию-фильтр 8](#_Toc499823784)

[4. Получить информацию об исключении с помощью GetExceptionInformation. Генерировать исключение с помощью RaiseException 8](#_Toc499823785)

[5. Использовать функции UnhandleExceptionFilter и SetUnhandleExceptionFilter для необработанных исключений 12](#_Toc499823786)

[6. Обработать вложенные исключения 13](#_Toc499823787)

[7. Выйти из блока \_try с помощью goto и leave 15](#_Toc499823788)

[8. Преобразовать структурные исключения (\_try) в исключения языка С. Для этого использовать функцию-транслятор (translator) 16](#_Toc499823789)

[9. Использовать финальный обработчик finally 17](#_Toc499823790)

[10. Проверить корректность выхода из блока try с помощью функции AbnormalTermination в финальном обработчике finally 18](#_Toc499823791)

[Вывод 19](#_Toc499823792)

1. **Задание**
2. Сгенерировать и обработать исключение с помощь WinAPI функций
3. Получить код исключения с помощью GetExceptionCode двумя способами:
   * использовать эту функцию в выражении фильтре;
   * использовать эту функцию в обработчике;
4. Создать собственную функцию-фильтр.
5. Получить информацию об исключении с помощью GetExceptionInformation. генерировать исключение с помощью RaiseException.
6. Использовать функции InhandleExceptionFilter и SetUnhandleExceptionFilter для необработанных исключений.
7. Обработать вложенные исключения.
8. Выйти из блока \_try с помощью:

* goto;
* leave.

1. Преобразовать структурные исключения (\_try) в исключение языка С. Для этого использовать функцию-транслятор (translator).
2. Использовать финальный обработчик finally.
3. Проверить корректность выхода из блока \_try с помощью функции AbnormalTermination в финальном обработчике finally.
4. **Решение**
5. **Сгенерировать и обработать исключение с помощь WinAPI функций**

Исключение — это событие при выполнении программы, которое приводит к её ненормальному или неправильному поведению. Для того, чтобы выполнение программы продолжилось далее необходимо или аварийное завершение программы, или немедленное её восстановление.

Для операционных системах Windows имеется специальный механизм структурной обработки исключений (structured exception handling). SEH обеспечивает надежность программ, благодаря ему приложения получают возможность реагировать на такие непредсказуемые события, как исключения адресации, арифметические сбои и системные ошибки. Использование SEH позволяет программам осуществлять корректный выход из любой точки программного блока и автоматически выполнять предусмотренную программистом обработку ошибок для восстановления своей работоспособности.

Принцип работы SEH: в программе место, в котором может произойти исключение, выделяется отдельным блоком кода (называют его фреймом).  После фрейма ниже записывают программный блок, который обрабатывает происшедшее исключение (обработчик исключения). По завершению обработки исключения код продолжается далее.

Для этого в C# предназначена конструкция try...catch...finally. Те инструкции программы, где ожидается возможность появления исключительных ситуаций, содержатся в бло­ке try. Если в блоке try возникает исключение, т. е. ошибка, то генерируется исключение. Исклю­чение перехватывается, используя catch, и обрабатывается. Код в блоке finally выполняется даже в том случае, если создано исключение. Далее представлен код 2-х исключений на C#.

Исключение EXCEPTION\_INT\_DIVIDE\_BY\_ZERO (на языке С# DivideByZeroException):

static void Exception\_DivideByZeroException()

{

int result;

int a = 10;

int b = Int16.Parse(Console.ReadLine());

try

{

result = a / b;

Console.WriteLine("{0} / {1} = {2:0.##}", a, b, result);

}

catch (DivideByZeroException e)

{

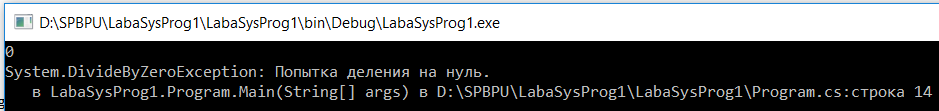
Console.WriteLine(e);

}

Console.ReadKey();

}

Результат при вводе значения “0”:



Исключение IndexOutOfRangeException:

static void Exception\_IndexOutOfRangeException()

{

int[] mass = new int[3] { 0, 1, 2 };

int i = Int16.Parse(Console.ReadLine());

try

{

Console.WriteLine(mass[i]);

}

catch (IndexOutOfRangeException e)

{

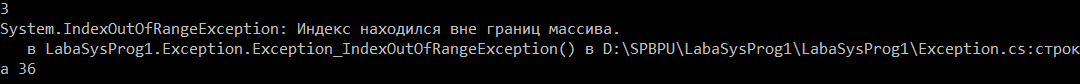
Console.WriteLine(e);

}

Console.ReadKey();

}

Результат при вводе значения “3”:



1. **Получить код исключения с помощью GetExceptionCode двумя способами**
   * 1. использовать эту функцию в обработчике.
     2. использовать эту функцию в выражении фильтре.

Функция GetExceptionCode может вызываться только в выражении-фильтре или в блоке обработки исключения. Следовательно, эта функция вызывается всегда только в том случае, если исключение произошло. В языке C# функция GetExceptionCode находиться в классе Marshal.

Назначение функции GetExceptionCode:

* В выражении фильтра она используется для того, чтобы определить выполняет ли текущий обработчик исключения обработку исключений с данным кодом или нужно продолжить поиск подходящего обработчика исключения.
* В блоке обработки исключения она также предназначена для проверки кодов исключений, которые обрабатывает текущий обработчик исключения, но в этом случае поиск другого обработчика исключений не выполняется.

1. Функция GetExceptionCode в обработчике:

static void Exception\_in\_handler()

{

int[] mass = new int[3] { 0, 1, 2 };

int i = Int16.Parse(Console.ReadLine());

int divider=Int16.Parse(Console.ReadLine());

try

{

Console.WriteLine(mass[i]/divider);

}

catch (System.Exception)

{

Console.WriteLine(Marshal.GetExceptionCode());

}

Console.ReadKey();

}

Чтобы использовать Marshal.GetExceptionCode() для вывода номера ошибки необходимо включить библиотеку System.Runtime.InteropServices.

Результат при вводе значения “2” и “0” (попадает под исключение DivideByZeroException):



Результат при вводе значения “3” и “1” (попадает под исключение IndexOutOfRangeException):



1. Функция GetExceptionCode в выражении фильтре:

static void Exception\_in\_filter()

{

int[] mass = new int[3] { 0, 1, 2 };

int i = Int16.Parse(Console.ReadLine());

int divider = Int16.Parse(Console.ReadLine());

try

{

Console.WriteLine(mass[i] / divider);

}

catch (System.Exception) when (divider==0)

{

Console.WriteLine("divider=0");

Console.WriteLine("Error: -1073741676");

}

catch (System.Exception) when (i>2)

{

Console.WriteLine("i>2");

Console.WriteLine("Error: -532462766");

}

catch (System.Exception)

{

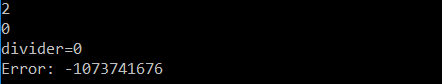
Console.WriteLine(Marshal.GetExceptionCode());

}

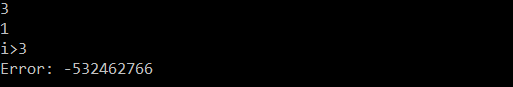
Console.ReadKey();

}

Результат при вводе значения “2” и “0” (попадает под исключение DivideByZeroException):



Результат при вводе значения “3” и “1” (попадает под исключение IndexOutOfRangeException):



Если будет исключение, не вошедшее в условия divider=0 и i>3, программа зайдет в последний catch и выдаст просто номер ошибки.

1. **Создать собственную функцию-фильтр**

Можно использовать так называемую функцию-фильтр, если для принятия решения об обработке исключения требуется более детально обработать информацию об исключении. Функции GetExceptionCode и GetExceptionlnformation в функции фильтра вызывать не разрешается, хотя эти функции могут вызываться для инициализации параметров функции фильтра. Но это работает только на C и C++. В C# можно это заменить использованием комбинации if и throw (Оператор throw используется для сообщения о случаях аномальных ситуаций (исключений) в ходе выполнения программы).

Далее рассмотрим такой пример в C# для исключений DivideByZeroException и IndexOutOfRangeException:

static void Main(string[] args)

{

int[] mass = new int[3] { 0, 1, 2 };

int i = Int16.Parse(Console.ReadLine());

int divider = Int16.Parse(Console.ReadLine());

try

{

Console.WriteLine(Metod(mass,divider,i));

}

catch (DivideByZeroException)

{

Console.WriteLine("Error! divider=0");

}

catch (IndexOutOfRangeException)

{

Console.WriteLine("Error! i>2");

}

Console.ReadKey();

}

static public int Metod (int[] mass, int divider, int i)

{

if (divider==0)

throw new DivideByZeroException();

if (i > 2)

throw new IndexOutOfRangeException();

return mass[i] / divider;

}

1. **Получить информацию об исключении с помощью GetExceptionInformation. Генерировать исключение с помощью RaiseException**

Так как язык С# более высокоуровневый (нежели С++ и С), у него нет таких функций, какGetExceptionInformation и RaiseException. Поэтому далее мы перейдем на язык C++.

Более подробную информацию об исключении можно получить при помощи вызова функции GetExceptionlnformation, которая имеет следующий прототип:

LPEXCEPTION\_POINTERS GetExceptionlnformation(void);

Возвращаемая структура содержит два указателя на структуры: Context и Exception\_Record.

1. В структуру типа Context система записывает содержимое всех регистров процессора на момент исключения. Эта структура имеет довольно громоздкое описание, которое можно найти в заголовочном файле WinNt.h.
2. В структуру Exception\_Record система записывает информацию об исключении. Структура имеет следующий формат:

typedef struct EXCEPTION\_RECORD {

DWORD ExceptionCode;

DWORD ExceptionFlags,

strict \_EXCEPTION\_RECORD \*ExceptionRecord;

PVOID ExceptionAddress,

DWORD Number Parameters,

ULONG\_PTR Exceptionlnformation [ EXCEPTION\_MAXIMUM\_PARAMETERS ] ;

} EXCEPTION\_RECORD, \*PEXCEPTION\_RECORD;

Поля этой структуры имеют следующее назначение:

* ExceptionCode содержит код исключения, который может принимать такие же значения, как и код исключения, возвращаемый функцией GetExceptionCode.
* ExceptionFlags может принимать одно из двух значений:
* 0 — обозначает, что после обработки исключения возможно возобновление выполнения программы;
* EXCEPTION\_NONCONTINUABLE — обозначает, что после обработки исключения возобновление выполнения программы невозможно. Если это значение установлено и выполнена попытка возобновления выполнения программы, то система выдаст исключение EXCEPTION\_NONCONTINUABLE\_EXCEPTION.
* ExceptionRecord содержит указатель на следующую структуру типа exception\_record, которая может быть создана в случае вложенных исключений.
* ExceptionAddress содержит адрес инструкции в программе, на которой произошло исключение.
* NumberParameters содержит количество параметров, заданных в поле Exceptioninformation, которое является последним в этой структуре.

Поле Exceptionlnformation[EXCEPTION\_MAXIMUM\_PARAMETERS] Определяет массив 32-битных аргументов, которые описывают исключение. Элементы этого массива могут использоваться функцией генерации программных исключений RaiseException.

Сделаем важное замечание о том, что функция GetExceptionlnformation может вызываться только в выражении фильтра. Поэтому эта функция вызывается всегда только в том случае, если исключение произошло. Кроме того, структуры типа exception\_pointers, exception\_record и context действительны только на время вычисления выражения-фильтра. Чтобы использовать содержимое структур типа exception\_record и context в блоке обработки исключения, его нужно сохранить в объявленных в программе переменных такого же типа.

Как видно из описания структуры EXCEPTION\_RECORD, функцию GetExceptionlnformation можно использовать для двух целей:

1. Получение более подробной информации об исключении, учитывая содержимое структуры типа context;
2. Обработка вложенных исключений.

Ниже приведены программы, которые выводят на консоль информацию об исключении, используя для получения этой информации функцию GetExceptionlnformation.

Исключение EXCEPTION\_INT\_DIVIDE\_BY\_ZERO

#include "stdafx.h"

#include <windows.h>

#include <iostream>

using namespace std;

EXCEPTION\_RECORD er; // информация об исключении

DWORD filter\_function(EXCEPTION\_POINTERS \*p)

{

// сохраняем содержимое структуры EXCEPTION\_RECORD

er = \*(p->ExceptionRecord);

return EXCEPTION\_EXECUTE\_HANDLER;

}

int main()

{

\_try

{

RaiseException(

EXCEPTION\_INT\_DIVIDE\_BY\_ZERO,

0, // continuable exception

0, NULL); // no arguments

}

\_except(filter\_function(GetExceptionInformation()))

{

// распечатываем информацию об исключении

cout << "ExceptionCode = " << er.ExceptionCode << endl;

cout << "ExceptionFlags = " << er.ExceptionFlags << endl;

cout << "ExceptionRecord = " << er.ExceptionRecord << endl;

cout << "ExceptionAddress = " << er.ExceptionAddress << endl;

cout << "NumberParameters = " << er.NumberParameters << endl;

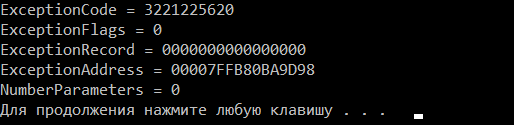
}

system("pause");

return 0;

}

Результат выполнения программы:



#include "stdafx.h"

#include <windows.h>

#include <iostream>

using namespace std;

EXCEPTION\_RECORD er; // информация об исключении

DWORD filter\_function(EXCEPTION\_POINTERS \*p)

{

// сохраняем содержимое структуры EXCEPTION\_RECORD

er = \*(p->ExceptionRecord);

return EXCEPTION\_EXECUTE\_HANDLER;

}

int main()

{

\_try

{

RaiseException(

EXCEPTION\_ARRAY\_BOUNDS\_EXCEEDED,

0, // continuable exception

0, NULL); // no arguments

}

\_except(filter\_function(GetExceptionInformation()))

{

// распечатываем информацию об исключении

cout << "ExceptionCode = " << er.ExceptionCode << endl;

cout << "ExceptionFlags = " << er.ExceptionFlags << endl;

cout << "ExceptionRecord = " << er.ExceptionRecord << endl;

cout << "ExceptionAddress = " << er.ExceptionAddress << endl;

cout << "NumberParameters = " << er.NumberParameters << endl;

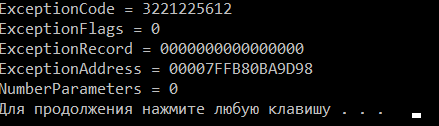
}

system("pause");

return 0;

}

Результат выполнения программы:



1. **Использовать функции UnhandleExceptionFilter и SetUnhandleExceptionFilter для необработанных исключений**

UnhandledExceptionFilter

Может случиться, что программа выкинет исключение, на которое обработчика в программе нет. В этом случае вызывается функция-фильтр системного обработчика исключений, которая выводит на экран сообщение с предложением пользователю аварийного завершения программы или выполнения отладки.

Системная функция-фильтр UnhandledExceptionFilter имеет следующий прототип:

LONG UnhandledExceptionFilter(PEXCEPTION\_POINTERS pExceptionlnfo);

Функция имеет 1 параметр, который указывает на структуру типа EXCEPTION\_INFO и возвращает одно из следующих значений:

1. EXCEPTION\_CONTHSTUE\_SEARCH — передать управление отладчику приложения;
2. EXCEPTION\_EXECUTE\_HANDLER —передать управление обработчику исключений.

SetUnhandledExceptionFilter

Приложение может заменить системную функцию-фильтр с помощью функции SetUnhandledExceptionFilter, которая имеет следующий прототип:

LPTOP\_LEVEL\_EXCEPTION\_FILTER SetUnhandledExceptionFilter(

LPTOP\_LEVEL\_EXCEPTION\_FILTER lpTopLevelExceptionFilter);

В данной функции происходит возвращения адреса старой функции фильтра или же значения null, если установлен системный обработчик исключений.

Функция имеет 1 параметр, который является указателем на новую функцию-фильтр, устанавливаемая вместо системной. Эта функция-фильтр должна иметь прототип, соответствующий системной функции фильтра UnhandledExceptionFilter, и возвращать одно из следующих значений:

1. EXCEPTION\_EXECUTE\_HANDLER — выполнение программы прекращается;
2. EXCEPTION\_CONTINUE\_EXECUTION — возобновить исполнение программы с точки исключения;
3. EXCEPTION\_CONTINUE\_SEARCH — выполняется системная функция UnhandledExceptionFilter.

Для того чтобы восстановить системную функцию-фильтр UnhandledExceptionFilter, нужно вызвать функцию UnhandledExceptionFilter с параметром null.

Ниже следует пример программы, которая устанавливает новую функцию-фильтр для необработанных исключений, а затем восстанавливает системную функцию UnhandledExceptionFilter.

#include "stdafx.h"

#include <windows.h>

#include <iostream>

using namespace std;

LONG filterNew(PEXCEPTION\_POINTERS pExceptionInfo)

{

cout << "New filter-function is called." << endl;

cout << "Exception code = " << hex << pExceptionInfo->ExceptionRecord->ExceptionCode << endl;

return EXCEPTION\_EXECUTE\_HANDLER;

}

int main()

{

LPTOP\_LEVEL\_EXCEPTION\_FILTER filterOld;

filterOld = SetUnhandledExceptionFilter((LPTOP\_LEVEL\_EXCEPTION\_FILTER)filterNew);

// установили новую функцию-фильтр необраб. исключений filterOld

cout << "Old filter-function address = " << hex << filterOld << endl;

// вывели адрес старой функции-фильтра

cout << "New filter-function address = " << hex << filterNew << endl;

// вывели адрес новой функции-фильтра

// hex - ввод/вывод в шестнадцатеричной системе счисления

/\*RaiseException(

EXCEPTION\_ARRAY\_BOUNDS\_EXCEEDED,

0,

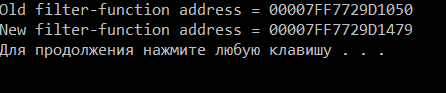
0, NULL);

system("pause");\*/

return 0x0;

}

Результат выполнения программы:



1. **Обработать вложенные исключения**

Архитектура обработки исключений в WinAPI позволяет обрабатывать вложенные блоки \_\_try, \_\_except.

В этом случае если функция-фильтр внутреннего блока except возвращает значение EXCEPTION\_CONTINUE\_SEARCH, то система удаляет все локальные объекты, принадлежащие текущим блокам try и except, и продолжает поиск обработчика исключений во внешних блоках try и except.

Напишем программу, у которой в \_try вложены ещё 2 проверки исключений. Пусть первую проверку на исключение EXCEPTION\_ARRAY\_BOUNDS\_EXCEEDED программа пройдет в созданный для него \_except, а вторую проверку на исключение EXCEPTION\_INT\_DIVIDE\_BY\_ZERO мы не будем ловить в этом блоке. Поэтому это исключение выйдет из вложенного \_try и пойдет в основной \_try, где для этого исключения есть обработчик. Ниже представлен код программы.

#include "stdafx.h"

#include <windows.h>

#include <iostream>

using namespace std;

int main()

{

\_try

{

\_try

{

RaiseException(EXCEPTION\_ARRAY\_BOUNDS\_EXCEEDED, 0, 0, NULL);

}

\_except(GetExceptionCode() == EXCEPTION\_ARRAY\_BOUNDS\_EXCEEDED ?

EXCEPTION\_EXECUTE\_HANDLER : EXCEPTION\_CONTINUE\_SEARCH)

{

cout << "Exception: EXCEPTION\_ARRAY\_BOUNDS\_EXCEEDED" << endl;

}

\_try

{

RaiseException(EXCEPTION\_INT\_DIVIDE\_BY\_ZERO, 0, 0, NULL);

}

\_except(GetExceptionCode() == EXCEPTION\_ARRAY\_BOUNDS\_EXCEEDED ?

EXCEPTION\_EXECUTE\_HANDLER : EXCEPTION\_CONTINUE\_SEARCH)

{

cout << "Exception: EXCEPTION\_ARRAY\_BOUNDS\_EXCEEDED" << endl;

}

}

\_except(GetExceptionCode() == EXCEPTION\_INT\_DIVIDE\_BY\_ZERO ?

EXCEPTION\_EXECUTE\_HANDLER : EXCEPTION\_CONTINUE\_SEARCH)

{

cout << "Exception: EXCEPTION\_INT\_DIVIDE\_BY\_ZERO" << endl;

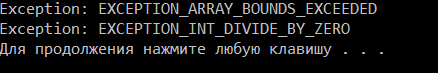
}

system("pause");

return 0;

}

Результат выполнения программы:



1. **Выйти из блока \_try с помощью goto и leave**

Ключевое слово \_\_leave может использоваться только в защищенном разделе оператора try-except. Оно приводит к тому, что выполнение переходит в конец защищенного раздела. Выполнение продолжается с первого оператора в обработчике завершения.

Оператор goto может также выйти из защищенного раздела, однако в этом случае производительность снизится, поскольку будет вызвано освобождение стека. Оператор \_\_leave более эффективен, поскольку не вызывает освобождение стека.

Напишем программу, в которой мы будем вводить некоторое число, на которое в последствии будет делиться число 100. Если оно будет не равно нулю, то программы выйдет через goto по ссылке exit\_try. Проверить мы это можем подстановкой вывода записи "Exited from \_except" перед концом программы. Если она не выведется, то значит программа прошло через goto на конец программы. Если же число будет равно 0, то ловим ошибку с помощью EXCEPTION\_EXECUTE\_HANDLER.

#include "stdafx.h"

#include <windows.h>

#include <iostream>

using namespace std;

int main()

{

int a=100;

int b;

cout <<"a=100/b. Enter number b: ";

cin >> b;

\_try

{

if (b != 0)

{

cout << "a/b=" << (a / b) << endl;

cout << "exit through \_goto" << endl;

goto exit\_try; // выходим их блока

}

else

a = a / b;

}

\_except(EXCEPTION\_EXECUTE\_HANDLER)

{

cout << "Finded exception: EXCEPTION\_INT\_DIVIDE\_BY\_ZERO" << endl;

}

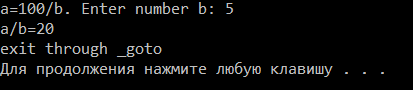
cout << "Exited from \_except" << endl;

exit\_try: system("pause");

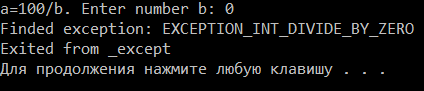
return 0;

}

Результат вывода программы при вводе значения 5:



Результат вывода программы при вводе значения 0:



Далее посмотрим на ключевое слово \_leave. С помощью него просто при значении “0” заведомо выйдем из \_try.

#include "stdafx.h"

#include <windows.h>

#include <iostream>

using namespace std;

int main()

{

int a=100;

int b;

cout <<"a=100/b. Enter number b: ";

cin >> b;

\_try

{

if (b == 0)

{

cout << "Exited throught \_leave"<< endl;

\_leave;

}

cout << "a/b=" << (a / b) << endl;

}

\_except(EXCEPTION\_EXECUTE\_HANDLER)

{

cout << "Finded exception: EXCEPTION\_INT\_DIVIDE\_BY\_ZERO" << endl;

}

system("pause");

return 0;

}

Результат вывода программы при вводе значения 0:



1. **Преобразовать структурные исключения (\_try) в исключения языка С. Для этого использовать функцию-транслятор (translator)**

В C++ существует функция \_set\_se\_transiator. Она обрабатывает исключения Win32 (структурированные исключения C) как типизированные исключения C++. Для работы с функцией необходимо подключить библиотеку eh.h.

Синтаксис:

\_se\_translator\_function \_set\_se\_translator (\_se\_translator\_function seTransFunction);

Параметр seTransFunction – это указатель на функцию-преобразователь структурированного исключения языка С, создаваемую пользователем.

Пример использования функции \_set\_se\_transiator для исключения EXCEPTION\_INT\_DIVIDE\_BY\_ZERO:

#include <windows.h>

#include <iostream>

#include <eh.h>

using namespace std;

void se\_trans\_func(unsigned code, EXCEPTION\_POINTERS \*)

{

throw code;

}

int main()

{

int a = 10, b = 0;

// устанавливаем функцию-транслятор

\_set\_se\_translator(se\_trans\_func);

// перехватываем структурное исключение средствами C++

try

{

a /= b; // ошибка, деление на ноль

}

catch (unsigned code)

{

cout << "Exception code = " << hex << code << endl;

}

return 0;

}

1. **Использовать финальный обработчик finally**

Кроме try и catch, существует также ключевое слово – finally. При любой передаче управления из блока try, независимо от того, произошло или нет исключение внутри этого блока, предварительно управление будет передано блоку finally. Такой способ обработки исключений называется финальная обработка исключений. Финальная обработка исключений используется для того, чтобы при любом исходе исполнения блока try освободить ресурсы, которые были захвачены внутри этого блока. Такими ресурсами могут быть память, файлы, критические секции и т. д.

Структурно финальная обработка исключений выглядит следующим образом:

\_try

{

// охраняемый код

}

finally

{

// финальный код

}

Напишем простую программу с ключевым словом finally:

#include "stdafx.h"

#include <Windows.h>

#include <iostream>

using namespace std;

int main()

{

\_try

{

cout << "The try block finished." << endl;

}

\_finally

{

cout << "The finally block finished." << endl;

}

system("pause");

return 0;

}

Результат вывода программы:



1. **Проверить корректность выхода из блока try с помощью функции AbnormalTermination в финальном обработчике finally**

Управление из блока try может быть передано одним из следующих способов:

* нормальное завершение блока;
* выход из блока при помощи управляющей инструкции leave;
* выход из блока при помощи одной из управляющих инструкций return, break, continue или goto языка программирования C++;
* передача управления обработчику исключения.

В первых двух случаях считается, что блок try завершился нормально, а в последних двух случаях — ненормально. Для того чтобы определить, как завершился блок try, используется функция AbnormaiTermination, которая имеет следующий прототип:

BOOL AbnormaiTermination (VOID) ;

В случае если блок try завершился ненормально, эта функция возвращает ненулевое значение, а в противном случае — значение false. Используя функцию AbnormaiTermination, ресурсы, захваченные в блоке try, можно освобождать только в том случае, если блок try завершился ненормально.

Ниже представлена программа, которая использует функцию AbnormaiTermination для проверки нормального завершения блока try.

Пример использования функции AbnormaiTermination для исключения EXCEPTION\_INT\_DIVIDE\_BY\_ZERO:

#include "stdafx.h"

#include <windows.h>

#include <iostream>

#include <eh.h>

using namespace std;

int main()

{

int a = 10;

int b = 0;

\_try

{

a = a / b;

}

\_finally

{

// если блок try закончился нормально

if (!AbnormalTermination())

{

cout << "try was done." << endl;

}

else

// иначе нечего освобождать

cout << "try wasn't done." << endl;

}

return 0;

}

# Вывод

Существуют небольшие отличия механизма структурной обработки исключений в Windows от механизма обработки исключений, принятого в языке программирования C++.

Это связано с тем, что:

* механизм структурной обработки исключений был разработан раньше, чем принят стандарт языка C++;
* механизм структурной обработки исключений в отличие от языка программирования C++ ориентирован не только на обработку программных исключений, но и на обработку аппаратных исключений.

В SEH исключение рассматривается как ошибка, происшедшая при выполнении программы. В языке программирования C++ используется более абстрактный подход и исключение рассматривается как объект произвольного типа, который может выбросить программа, используя оператор throw.

В свою очередь обработчик исключения catch может рассматриваться как функция с одним параметром, которая выполняется только в том случае, если тип ее параметра соответствует типу выброшенного исключения.