**Федеральное государственное автономное образовательное учреждение**

**Высшего профессионального образования**

Санкт-Петербургский политехнический университет

Институт информационных технологий и управления

Кафедра компьютерных систем и программных технологий

**Отчёт по лабораторной работе №2**

«Исключения в Windows»

Работу выполнил студент гр. № 53501/3 Цыганов А.А.

Работу принял преподаватель \_\_\_\_\_\_\_\_\_\_\_\_\_\_ Душутина Е.В.

Санкт-Петербург

2016

# Цель работы:

Рассмотреть способы обработки исключений в ОС Windows:

1. Сгенерировать и обработать исключения с помощью функций WinAPI;

2. Получить код исключения с помощью функции GetExceptionCode.

• Использовать эту функции в выражении фильтре;

• Использовать эту функцию в обработчике.

3. Создать собственную функцию-фильтр;

4. Получить информацию об исключении с помощью функции GetExceptionInformation; сгенерировать исключение с помощью функции RaiseException;

5. Использовать функции UnhandleExceptionFilter и Set UnhandleExceptionFilter для необработанных исключений;

6. Обработать вложенные исключения;

7. Выйти из блока \_\_try с помощью оператора goto;

8. Выйти из блока \_\_try с помощью оператора leave;

9. Преобразовать структурное исключение в исключение языка С, используя функцию translator;

10. Использовать финальный обработчик finally;

1. Проверить корректность выхода из блока \_\_try с помощью функции AbnormalTermination в финальном обработчике finally.

# Ход работы:

Обработка исключений в ОС Windows возможна средствами языка С++ и средствами Windows (SEH) Structured Exception Handling. В программе выделяется блок кода – фрейм, в котором может произойти исключение (охраняемый код), за фреймом размещается код обработчика.

В SEH исключение – это ошибка при выполнении программы. В SEH пригоден как для программных, так и для аппаратных исключений, в отличие от средств С++.

Структура программного фрагмента обработки исключения SEH:

\_try { /\*охраняемый код\*/ }

\_except (выражение-фильтр) { /\*код обработки исключения\*/ }

Значения (выражения-фильтра):

EXCEPTION\_EXECUTE\_HANDLER – управление передается обработчику

EXCEPTION\_CONTINUE\_SEARCH – продолжить поиск обработчика

EXCEPTION\_EXECUTE\_EXECUTION – управление передается в точку прерывания программы.

*GetExceptionCode(), GetExceptionInformation() –* функции для получения информации об исключении, которые используются в выражении-фильтре.

В С++ исключение - это объект произвольного типа, который может выбросить программа, используя оператор ***throw****.* Обработчик исключения ***catch*** может рассматриваться как функция с одним параметром, которая выполняется только при совпадении типа ее параметра с типом выброшенного исключения

В этой работе будут рассматриваться исключения:

* деление на ноль (EXCEPTION\_INT\_DIVIDE\_BY\_ZERO);
* возник сигнал о пошаговом выполнении программы (трассировочная ловушка или другой механизм) (EXCEPTION\_SINGLE\_STEP).

В случае, если ошибка генерируется, то программа передает управления соответствующему блоку обработчику, который будет выводит в консоль информацию об исключении.

# Сгенерировать и обработать исключения с помощью функций WinAPI

Ниже представлен код программы (Листинг 1), в котором приведено обработка исключения деления числа на 0. После запуска программы пользователь вводит знаменатель и затем происходит деление на введенное число. В зависимости от введенного значений в программе может вызываться исключение.

Отладка и обработка исключение проводится в среде разработки Microsoft Visual Studio Enterprise 2015.

*Листинг 1. Генерация и обработка исключения с помощью функций WinAPI*

#include "stdafx.h"

#include <iostream>

#include <windows.h>

using namespace std;

int main(int argc, char\* argv[])

{

INT32 b = 1;

cout << "Lab 1 \nInput number: ";

cin >> b;

\_\_try {

cout << "\_\_try 1 / " << b << endl;

int result = 1 / b;

}

\_\_except (EXCEPTION\_EXECUTE\_HANDLER) {

cout << "Devision exception handler" << endl;

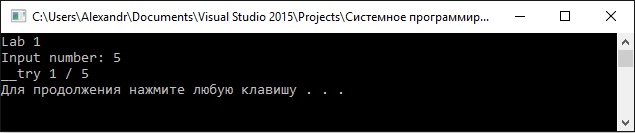
}

system("pause");

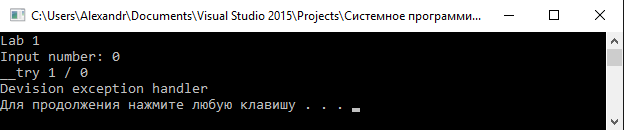
return 0;

}

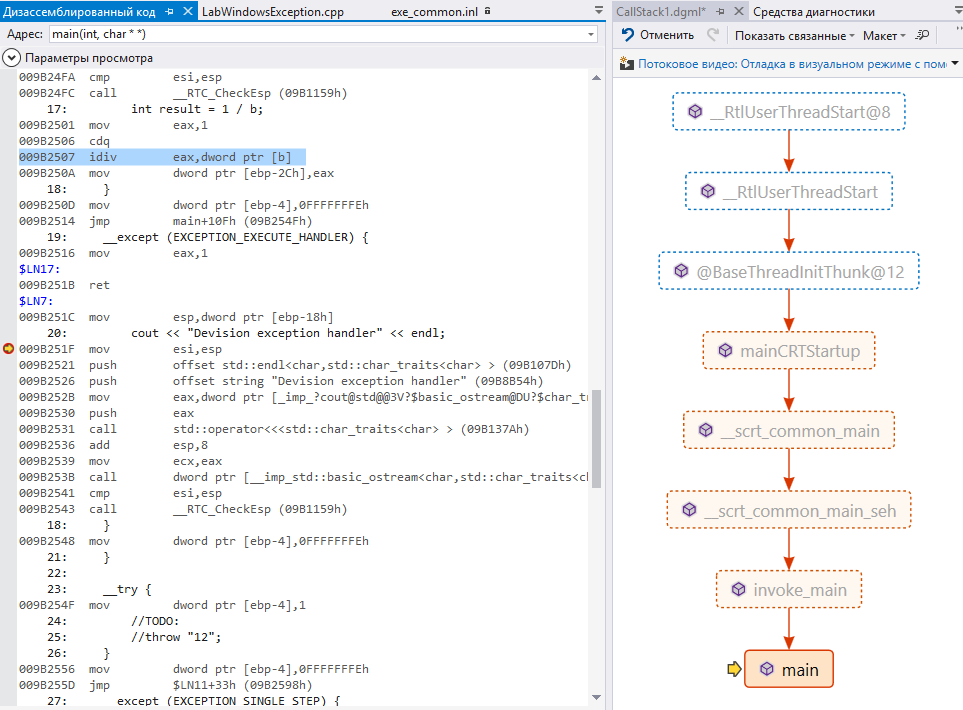
Ниже приведен результат выполнения программы без генерирования исключения, т.е. пользователь ввел число 5.

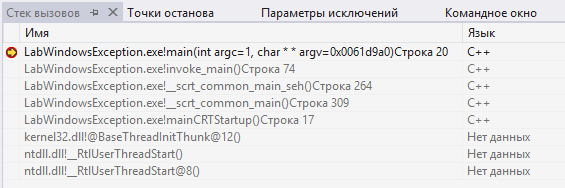


Если пользователь ввел 0, то происходит исключении в блоке \_\_try деление на ноль и управление переключается на блок \_\_except, в котором выводится информация в консоль.

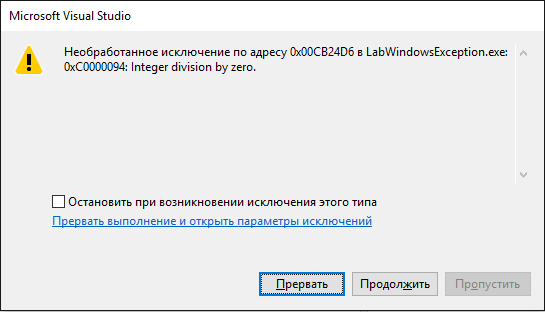


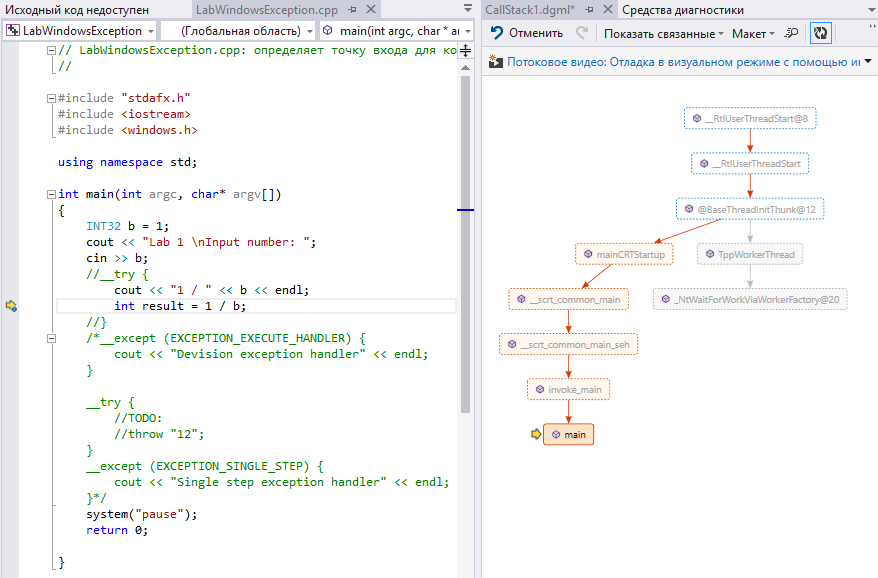
Visual Studio Enterprise позволяет визуализировать стек вызовов в любом потоке. Приведен дизассемблированный код, в котором виден переход на метку $LN7, т.к. в безопасном блоке \_\_try было получено исключение. Инструкции с 009B250А не были исполнены. В итоге программа завершилась успешно.

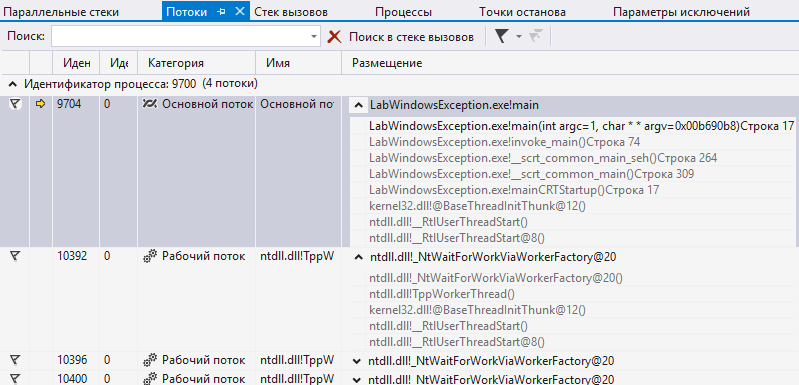




При условии, когда в коде программы деление на ноль происходит вне блока \_\_try и пользователь ввел 0, тогда Visual Studio приостановит программу, отладчик сообщает об ошибке. И укажет на место в коде, где произошло исключение. Ниже представлены стеки вызовов и необработанное исключение.







Далее проведем отладку программы в утилите WinDbg. Для этого откроем исполняемый файл LabWindowsException.exe и в консоли выполним gh (отладка с обработкой исключений). Ниже приведена консоль с найденным исключением (деление на ноль) с ассемблерной инструкцией.

Microsoft (R) Windows Debugger Version 10.0.10586.567 X86

Copyright (c) Microsoft Corporation. All rights reserved.

CommandLine: "C:\Users\Alexandr\Documents\Visual Studio 2015\Projects\Системное программирование\LabWindowsException\Debug\LabWindowsException.exe"

Symbol search path is: srv\*

Executable search path is:

ModLoad: 00ca0000 00cc2000 LabWindowsException.exe

ModLoad: 76fe0000 7715b000 ntdll.dll

ModLoad: 748e0000 749c0000 C:\WINDOWS\SysWOW64\KERNEL32.DLL

ModLoad: 75320000 7549e000 C:\WINDOWS\SysWOW64\KERNELBASE.dll

ModLoad: 588b0000 588cc000 C:\WINDOWS\SysWOW64\VCRUNTIME140D.dll

ModLoad: 587f0000 588a9000 C:\WINDOWS\SysWOW64\MSVCP140D.dll

ModLoad: 77320000 77493000 C:\WINDOWS\SysWOW64\ucrtbased.dll

ModLoad: 76e50000 76ecb000 C:\WINDOWS\SysWOW64\ADVAPI32.dll

ModLoad: 76d90000 76e4e000 C:\WINDOWS\SysWOW64\msvcrt.dll

ModLoad: 73e50000 73e94000 C:\WINDOWS\SysWOW64\sechost.dll

ModLoad: 74ce0000 74d8d000 C:\WINDOWS\SysWOW64\RPCRT4.dll

ModLoad: 73d10000 73d2e000 C:\WINDOWS\SysWOW64\SspiCli.dll

ModLoad: 73d00000 73d0a000 C:\WINDOWS\SysWOW64\CRYPTBASE.dll

ModLoad: 74300000 74358000 C:\WINDOWS\SysWOW64\bcryptPrimitives.dll

(2730.2834): Break instruction exception - code 80000003 (first chance)

eax=00000000 ebx=00000003 ecx=19800000 edx=00000000 esi=00ca00f8 edi=002e2000

eip=7708ccdc esp=0018f2b4 ebp=0018f2e0 iopl=0 nv up ei pl zr na pe nc

cs=0023 ss=002b ds=002b es=002b fs=0053 gs=002b efl=00000246

ntdll!LdrpDoDebuggerBreak+0x2b:

7708ccdc cc int 3

0:000> gh

**(2730.2834): Integer divide-by-zero - code c0000094 (first chance)**

First chance exceptions are reported before any exception handling.

This exception may be expected and handled.

\*\*\* WARNING: Unable to verify checksum for LabWindowsException.exe

eax=00000001 ebx=002e2000 ecx=db1d78e6 edx=00000000 esi=0018f694 edi=0018f690

eip=00cb24d6 esp=0018f694 ebp=0018f778 iopl=0 nv up ei pl zr na pe nc

cs=0023 ss=002b ds=002b es=002b fs=0053 gs=002b efl=00010246

LabWindowsException!main+0x96:

**00cb24d6 f77df8 idiv eax,dword ptr [ebp-8]** ss:002b:0018f770=00000000

0:000> !exchain

0018f7d4: LabWindowsException!\_except\_handler4+0 (00cb2970)

CRT scope 0, filter: LabWindowsException!\_\_scrt\_common\_main\_seh+1b0 (00cb2c80)

func: LabWindowsException!\_\_scrt\_common\_main\_seh+1cb (00cb2c9b)

0018f840: ntdll!\_except\_handler4+0 (7705ed70)

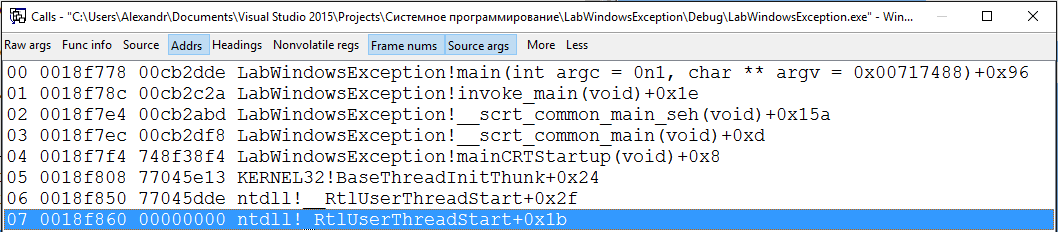
CRT scope 0, filter: ntdll!\_\_RtlUserThreadStart+3d7f9 (770835dd)

func: ntdll!\_\_RtlUserThreadStart+3d840 (77083624)

0018f858: ntdll!FinalExceptionHandlerPad58+0 (7706b80a)

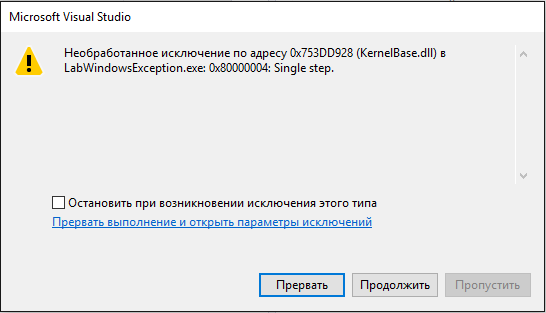
Invalid exception stack at ffffffff

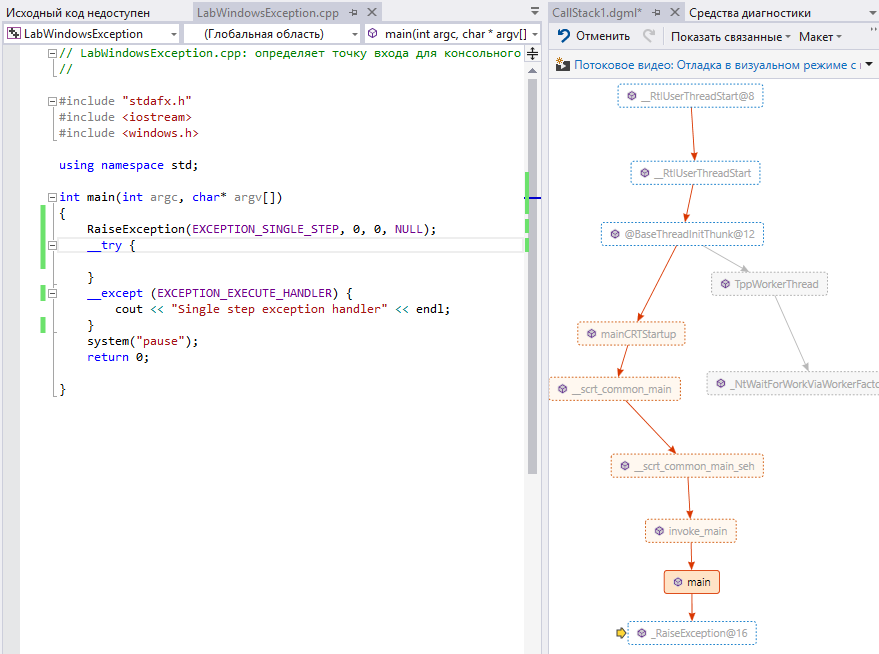
Ниже приведен стек вызовов:

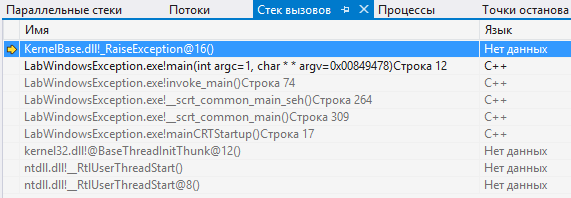


Полученные стеки вызовов в Visual Studio и WinDbg схожи между собой, и для дальнейшего выполнения работы будет использоваться отладчик Visual Studio, т. к. предоставляет более наглядное представление о состоянии программы в процессе отладки.

Согласно выбранному варианту необходимо обработать исключение: возник сигнал о пошаговом выполнении программы (трассировочная ловушка или другой механизм) (EXCEPTION\_SINGLE\_STEP). Значение EXCEPTION\_SINGLE\_STEP, значит отладочное событие было сгенерировано из-за пошагового режима. В интернете не удалось найти подходящее решение для генерации этого исключения. Поэту для примера было решено генерировать это исключение через функцию RaiseException(). Ниже приведено обработка этого исключения в Visual Studio:







# Получить код исключения с помощью функции GetExceptionCode

Функция GetExceptionCode используется для получения кода сгенерированного исключения. Функцию GetExceptionCode можно вызывать только из фильтра исключений, между скобками оператора \_except, или из обработчика исключений, т.к. именно в этот момент существуют структуры с контекстом исключения.

Ниже приведен код программы, в которой обработка исключения деления на ноль использует функцию GetExceptionCode в обработчике.

#include "stdafx.h"

#include <iostream>

#include <windows.h>

using namespace std;

int main(int argc, char\* argv[])

{

INT32 a = 1, b = 0;

\_\_try {

int result = a / b;

RaiseException(EXCEPTION\_SINGLE\_STEP, 0, 0, NULL);

}

\_\_except (EXCEPTION\_EXECUTE\_HANDLER) {

if (GetExceptionCode() == EXCEPTION\_SINGLE\_STEP)

cout << "Single step exception handler" << endl;

else if (GetExceptionCode() == EXCEPTION\_INT\_DIVIDE\_BY\_ZERO)

cout << "Devision exception handler" << endl;

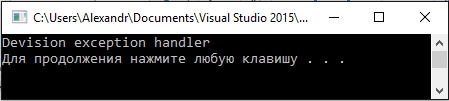
}

system("pause");

return 0;

}

После запуска программы происходит обработка исключения деление на ноль и в консоль выводится "Devision exception handler".



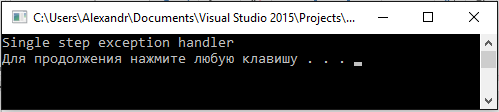
Далее изменим местами выполнение команд деление на ноль и пошаговая отладка. Затем после запуска программы в консоль выводится " Single step exception handler ", что означает функция GetExceptionCode() возвращает нужное значение для обработанного исключения.

\_\_try {

RaiseException(EXCEPTION\_SINGLE\_STEP, 0, 0, NULL);

int result = a / b;

}



Аналогичным образом работает программа, когда функцию GetExceptionCode использовать в блоке обработчика. Ниже приведен код программы с использованием функции GetExceptionCode в обработчике:

\_\_try

{

int result = a / b;

}

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

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

{

cout << "Devision exception handler" << endl;

}

\_\_try

{

int result = a / b;

}

\_\_except (GetExceptionCode() == EXCEPTION\_SINGLE\_STEP ?

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

{

cout << "Single step exception handler" << endl;

}

# Создать собственную функцию-фильтр

Ниже приведен код программы, в котором была создана своя собственная функция-фильтр. Внутри функции-фильтра происходит определения типа исключения и вызывающему обработчику возвращается указание об необходимости либо ненадобности обработки произошедшего исключения.

#include "stdafx.h"

#include <iostream>

#include <windows.h>

using namespace std;

LONG myFilter(DWORD dwExceptionGode) {

return dwExceptionGode == EXCEPTION\_INT\_DIVIDE\_BY\_ZERO ? EXCEPTION\_EXECUTE\_HANDLER

: EXCEPTION\_CONTINUE\_SEARCH;

}

int main(int argc, char\* argv[])

{

INT32 a = 1, b = 0;

\_\_try {

int result = a / b;

}

\_\_except (myFilter(GetExceptionCode())) {

cout << "Devision exception handler" << endl;

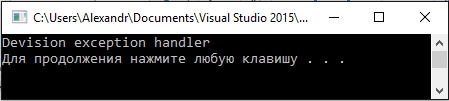
}

system("pause");

return 0;

}

Если в блоке \_\_try сгенерировать другое исключение, то его уже перехватит Visual Studio. Результат выполнения программы:



# Получить информацию об исключении с помощью функции GetExceptionInformation; сгенерировать исключение с помощью функции RaiseException;

RaiseException используется для принудительного генерирования программного исключения. Функция GetExceptionInformation позволяет получить информацию о контексте возникновения исключений.

Ниже приведена программа, которая использует функцию GetExceptionInformation для вывода информации об исключении (деление на ноль).

#include "stdafx.h"

#include <iostream>

#include <windows.h>

using namespace std;

LONG myFilter(DWORD dwExceptionGode, EXCEPTION\_POINTERS\* ep) {

return dwExceptionGode == EXCEPTION\_EXECUTE\_HANDLER;

}

int main(int argc, char\* argv[])

{

INT32 a = 1, b = 0;

\_\_try {

int result = a / b;

}

\_\_except (myFilter(GetExceptionCode(), GetExceptionInformation())) {

cout << "Single step exception handler" << endl;

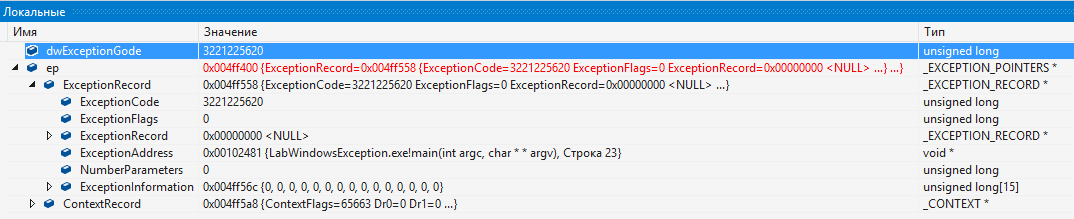
}

system("pause");

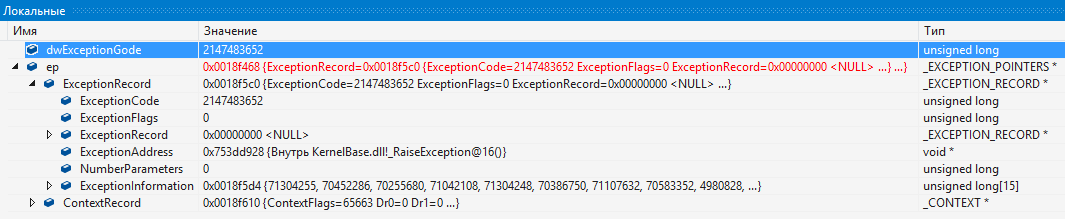
return 0;

}

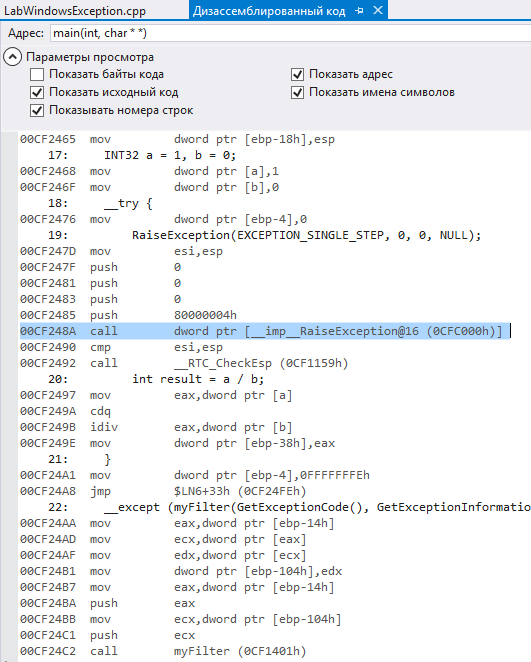
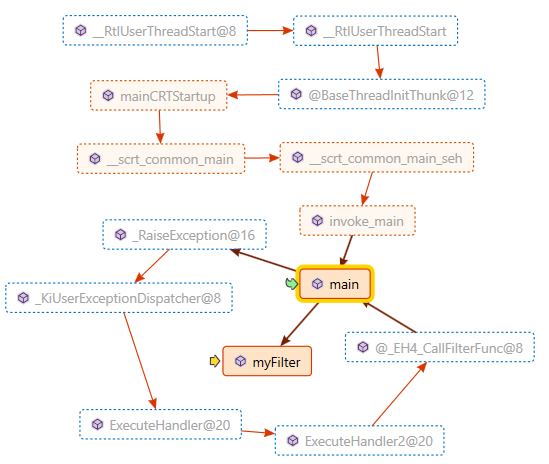
Внутри функции myFilter установлена точка остановки программы для просмотра содержимого переменной ep.

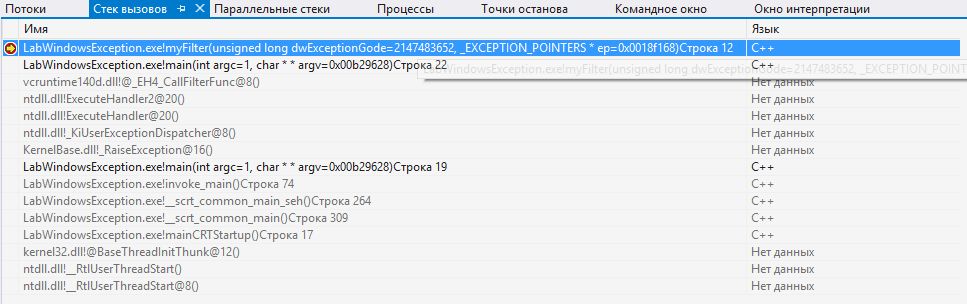


Для исключения EXCEPTION\_SINGLE\_STEP получены соответствующие данные:



При использовании функции для генерации исключения ниже приведен стек вызовов и дизассемблированный код:





# Использовать функции UnhandleExceptionFilter и Set UnhandleExceptionFilter для необработанных исключений

В Windows имеется возможность изменить фильтр, вызываемый по умолчанию, с помощью функции SetUnhandledExceptionFilter и реализовать собственный механизм обработки исключений.

#include "stdafx.h"

#include <iostream>

#include <windows.h>

using namespace std;

LONG WINAPI myUnhandledExceptionFilter(\_EXCEPTION\_POINTERS \*ep) {

printf("ep->ExceptionRecord->ExceptionAddress %p\n", ep->ExceptionRecord->ExceptionAddress);

printf("ep->ExceptionRecord->ExceptionCode %d\n", ep->ExceptionRecord->ExceptionCode);

printf("ep->ExceptionRecord->ExceptionFlags %d\n", ep->ExceptionRecord->ExceptionFlags);

printf("ep->ExceptionRecord->ExceptionInformation %ld\n", ep->ExceptionRecord->ExceptionInformation);

printf("ep->ExceptionRecord->ExceptionRecord %p\n", ep->ExceptionRecord->ExceptionRecord);

printf("ep->ExceptionRecord->NumberParameters %d\n", ep->ExceptionRecord->NumberParameters);

return EXCEPTION\_EXECUTE\_HANDLER;

}

int main(int argc, char\* argv[])

{

SetUnhandledExceptionFilter(myUnhandledExceptionFilter);

INT32 a = 1, b = 0;

\_\_try {

RaiseException(EXCEPTION\_SINGLE\_STEP, 0, 0, NULL);

int result = a / b;

}

\_\_except (EXCEPTION\_CONTINUE\_SEARCH) {

cout << "Single step exception handler" << endl;

}

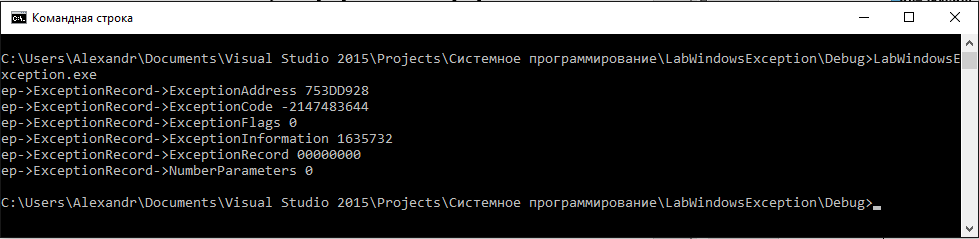
system("pause");

return 0;

}

Результат выполнения зависит от способа запуска программы. Если программа запущена под отладчиком, вызов функции SetUnhandledExceptionFilter будет проигнорирован и исключение будет перехвачено отладчиком.

Если запускать программу напрямую из консоли сработает пользовательский фильтр и выведется информация об исключении. В данном случае результатом будет «тихое» завершение приложения, без появления диалоговых окон.



# Обработать вложенные исключения

При вложенных исключениях происходит иерархический процесс обработки в блоках \_\_except. Если, при вложении одного \_\_try блока внутрь другого функция-фильтр внутреннего \_except возвращает EXCEPTION\_CONTINUE\_SEARCH, то система удаляет все локальные объекты, принадлежащие текущим блокам \_try и \_except и хранящиеся в стеке процесса, и продолжает поиск обработчика исключений во внешних блоках \_try и \_except.

Ниже приведен код программы в которой иерархически вложено два блока \_\_try с исключением деления на ноль.

#include "stdafx.h"

#include <iostream>

#include <windows.h>

using namespace std;

int main(int argc, char\* argv[])

{

INT32 a = 1, b = 0;

\_\_try {

\_\_try {

int result = a / b;

RaiseException(EXCEPTION\_SINGLE\_STEP, 0, 0, NULL);

}

\_\_except (GetExceptionCode() == EXCEPTION\_SINGLE\_STEP ?

EXCEPTION\_EXECUTE\_HANDLER : EXCEPTION\_CONTINUE\_SEARCH

) {

cout << "EXCEPTION\_SINGLE\_STEP exception handler" << endl;

}

}

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

EXCEPTION\_EXECUTE\_HANDLER : EXCEPTION\_CONTINUE\_SEARCH

) {

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

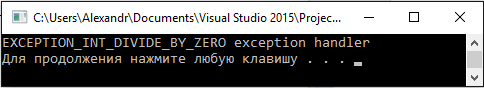
}

system("pause");

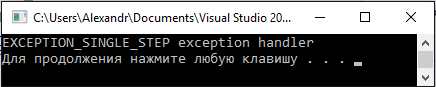
return 0;

}

После выполнения программы, в консоль выведется значение "EXCEPTION\_INT\_DIVIDE\_BY\_ZERO exception handler" и программа успешно завершится. При делении на ноль внутренний обработчик перехватывает управление, после чего управление передается инструкции, находящейся за последним обработчиком.



Если при делении не было исключения, то в консоль выведется значение "EXCEPTION\_SINGLE\_STEP exception handler" и программа успешно завершится. Т.е. произойдет раскрутка стека для поиска блока обработки требуемого исключения.



# Выйти из блока \_\_try с помощью оператора goto;

При использовании безусловного перехода (метки), программа мгновенно осуществляет переход, без раскручивания стека в блоке исключения. Ниже приведен текст программы и фрагмент дизассемблированного кода, в котором выделена инструкция перехода по метке.

#include "stdafx.h"

#include <iostream>

#include <windows.h>

using namespace std;

int main(int argc, char\* argv[])

{

INT32 a = 1, b = 0;

\_\_try {

**goto end;**

int result = a / b;

}

\_\_except (EXCEPTION\_EXECUTE\_HANDLER) {

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

}

**end:**

system("pause");

return 0;

}

12: INT32 a = 1, b = 0;

01105BE8 mov dword ptr [a],1

01105BEF mov dword ptr [b],0

13: \_\_try {

01105BF6 mov dword ptr [ebp-4],0

14: goto end;

01105BFD mov dword ptr [ebp-4],0FFFFFFFEh

**01105C04 jmp $LN7+33h (01105C52h)**

15: int result = a / b;

01105C06 mov eax,dword ptr [a]

01105C09 cdq

01105C0A idiv eax,dword ptr [b]

01105C0D mov dword ptr [ebp-38h],eax

16:

17: }

01105C10 mov dword ptr [ebp-4],0FFFFFFFEh

**01105C17 jmp $LN7+33h (01105C52h)**

18: \_\_except (EXCEPTION\_EXECUTE\_HANDLER) {

01105C19 mov eax,1

$LN10:

01105C1E ret

$LN7:

01105C1F mov esp,dword ptr [ebp-18h]

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

01105C22 mov esi,esp

01105C24 push offset std::endl<char,std::char\_traits<char> > (0110107Dh)

01105C29 push offset string "EXCEPTION\_INT\_DIVIDE\_BY\_ZERO exc"... (01108E90h)

01105C2E mov eax,dword ptr [\_imp\_?cout@std@@3V?$basic\_ostream@DU?$char\_traits@D@std@@@1@A (0110C098h)]

01105C33 push eax

01105C34 call std::operator<<<std::char\_traits<char> > (0110137Ah)

01105C39 add esp,8

01105C3C mov ecx,eax

01105C3E call dword ptr [\_\_imp\_std::basic\_ostream<char,std::char\_traits<char> >::operator<< (0110C0ACh)]

01105C44 cmp esi,esp

01105C46 call \_\_RTC\_CheckEsp (01101159h)

16:

17: }

01105C4B mov dword ptr [ebp-4],0FFFFFFFEh

20: }

21: end:

22: system("pause");

01105C52 mov esi,esp

20: }

21: end:

22: system("pause");

01105C54 push offset string "pause" (01108B58h)

01105C59 call dword ptr [\_\_imp\_\_system (0110C1CCh)]

01105C5F add esp,4

01105C62 cmp esi,esp

01105C64 call \_\_RTC\_CheckEsp (01101159h)

23: return 0;

01105C69 xor eax,eax

24: }

# Выйти из блока try с помощью оператора leave

Выход из блока \_\_try по метке \_\_leave осуществляется аналогичным образом. Ниже приведен текст программы и фрагмент дизассемблированного кода, в котором выделена инструкция перехода по метке. В отличие от goto – \_\_leave передает управление в конец блока \_try. После перехода выполняется обработчик завершения, что предполагает успешное завершение охраняемого блока.

#include "stdafx.h"

#include <iostream>

#include <windows.h>

using namespace std;

int main(int argc, char\* argv[])

{

INT32 a = 1, b = 0;

\_\_try {

\_\_leave;

int result = a / b;

}

\_\_except (EXCEPTION\_EXECUTE\_HANDLER) {

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

}

system("pause");

return 0;

}

12: INT32 a = 1, b = 0;

011B5BE8 mov dword ptr [a],1

011B5BEF mov dword ptr [b],0

13: \_\_try {

011B5BF6 mov dword ptr [ebp-4],0

14: \_\_leave;

**011B5BFD jmp main+69h (011B5C09h)**

15: int result = a / b;

011B5BFF mov eax,dword ptr [a]

011B5C02 cdq

011B5C03 idiv eax,dword ptr [b]

011B5C06 mov dword ptr [ebp-38h],eax

16:

17: }

011B5C09 mov dword ptr [ebp-4],0FFFFFFFEh

**011B5C10 jmp $LN6+33h (011B5C4Bh)**

18: \_\_except (EXCEPTION\_EXECUTE\_HANDLER) {

011B5C12 mov eax,1

$LN9:

011B5C17 ret

$LN6:

011B5C18 mov esp,dword ptr [ebp-18h]

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

011B5C1B mov esi,esp

011B5C1D push offset std::endl<char,std::char\_traits<char> > (011B107Dh)

011B5C22 push offset string "EXCEPTION\_INT\_DIVIDE\_BY\_ZERO exc"... (011B8E90h)

011B5C27 mov eax,dword ptr [\_imp\_?cout@std@@3V?$basic\_ostream@DU?$char\_traits@D@std@@@1@A (011BC098h)]

011B5C2C push eax

011B5C2D call std::operator<<<std::char\_traits<char> > (011B137Ah)

011B5C32 add esp,8

011B5C35 mov ecx,eax

011B5C37 call dword ptr [\_\_imp\_std::basic\_ostream<char,std::char\_traits<char> >::operator<< (011BC0ACh)]

011B5C3D cmp esi,esp

011B5C3F call \_\_RTC\_CheckEsp (011B1159h)

16:

17: }

011B5C44 mov dword ptr [ebp-4],0FFFFFFFEh

20: }

21: system("pause");

011B5C4B mov esi,esp

011B5C4D push offset string "pause" (011B8B58h)

20: }

21: system("pause");

011B5C52 call dword ptr [\_\_imp\_\_system (011BC1CCh)]

011B5C58 add esp,4

011B5C5B cmp esi,esp

011B5C5D call \_\_RTC\_CheckEsp (011B1159h)

22: return 0;

# Преобразовать структурное исключение в исключение языка С, используя функцию translator

Функция \_set\_se\_translator – это функция Visual C++, позволяющая преобразовывать SEH исключения С++ исключения. Для использования функции [\_set\_se\_translator](https://msdn.microsoft.com/ru-ru/library/5z4bw5h5.aspx) требуется указать параметр компилятора [/EHa](https://msdn.microsoft.com/ru-ru/library/1deeycx5.aspx) и не указывать параметр **/EHs**.

Сначала срабатывает транслятор, после чего происходит обработка С++ исключения. При генерации исключения деления на ноль, управление передается функции транслятору, в которой генерируется исключение С++ (throw runtime\_error("Divide by zero")), которое затем обрабатывается в catch-блоке. Ниже приведен код программы:

#include "stdafx.h"

#include <iostream>

#include <windows.h>

#include <eh.h>

using namespace std;

void myTranslator(unsigned int u, EXCEPTION\_POINTERS\* pExp)

{

cout << "myTranslator" << endl;

if (u == EXCEPTION\_INT\_DIVIDE\_BY\_ZERO)

throw runtime\_error("Divide by zero");

}

void myFunc()

{

INT32 a = 1, b = 0;

\_\_try

{

int result = a / b;

}

\_\_finally

{

cout << "In finally" << endl;

}

}

int main(int argc, char\* argv[])

{

try {

\_set\_se\_translator(myTranslator);

myFunc();

}

catch (runtime\_error re) {

cout << re.what() << endl;

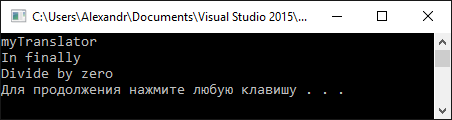
}

system("pause");

return 0;

}

Результат выполнения программы:



# Использовать финальный обработчик finally

Блок finally выполняется в любом случае, даже если исключение не было сгенерировано. С помощью финального блока можно организовать, например, гарантированное освобождение занимаемых ресурсов, вне зависимости от того была ли аварийная ситуация или нет. Ниже приведен код программы:

#include "stdafx.h"

#include <iostream>

#include <windows.h>

using namespace std;

int main(int argc, char\* argv[])

{

INT32 a = 1, b = 1;

\_\_try {

int result = a / b;

}

\_\_finally {

cout << "In finally" << endl;

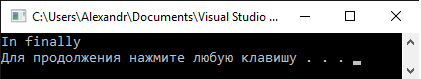
}

system("pause");

return 0;

}

Результат выполнения программы:



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

Функция AbnormalTermination позволяет определить состояние выполнения блока. Возвращается ненулевое значение, если завершение ненормальное, иначе – FALSE.

Ниже приведен код программы, в котором используются goto и leave в блоках \_\_try и соответствующие блоки \_\_finaly в которых выводится результат функции AbnormalTermination.

#include "stdafx.h"

#include <iostream>

#include <windows.h>

using namespace std;

int main(int argc, char\* argv[])

{

INT32 a = 1, b = 0;

\_\_try {

goto my;

int result = a / b;

}

\_\_finally {

if (AbnormalTermination())

cout << "Abnormal termination with goto." << endl;

else

cout << "Normal termination with goto." << endl;

}

my:

\_\_try {

\_\_leave;

RaiseException(EXCEPTION\_SINGLE\_STEP, 0, 0, NULL);

}

\_\_finally {

if (AbnormalTermination())

cout << "Abnormal termination with leave." << endl;

else

cout << "Normal termination with leave." << endl;

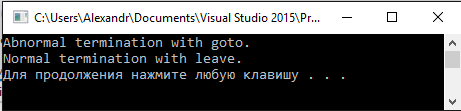
}

system("pause");

return 0;

}

После выполнения программы получены результаты. Использование goto является не нормальным, в отличие от оператора leave.



**Выводы:**

В ходе лабораторной работы были изучены основные методы работы с SEH и C++ исключениями. WinAPI предоставляет возможности для генерации SEH-исключений, выбора обработчика и, непосредственно, их обработки, а также взаимодействия с С++ - исключениями. Оператор **try-finally** является расширением Microsoft для языков C и С++ и позволяет целевым приложениям гарантировать выполнение кода очистки при прерывании выполнения блока кода. Очистка включает такие задачи, как отмена распределения памяти, закрытие файлов и освобождение их дескрипторов. Оператор **try-finally** особенно полезен для подпрограмм, в которых в нескольких местах выполняется проверка на наличие ошибок, способных вызвать преждевременное возвращение из подпрограммы.

Используемые источники

1. Душутина Е.В. Обработка исключений в ОС Windows. Фреймовая и финальная обработки.
2. Using an Exception Handler. https://msdn.microsoft.com/en-us/library/windows/desktop/ms681409(v=vs.85).aspx
3. GetExceptionCode macro. https://msdn.microsoft.com/en-us/library/windows/desktop/ms679356(v=vs.85).aspx
4. Знакомство с WinDBG – Часть 1. <http://www.securitylab.ru/analytics/453911.php>
5. Оператор try-finaly. https://msdn.microsoft.com/ru-ru/library/9xtt5hxz.aspx