

Отчет по расчетной работе № 1
по предмету «Системное программное обеспечение»

ОБРАБОТКА ИСКЛЮЧЕНИЙ В ОС WINDOWS

Работу выполнил студент гр. 53501/3 _____ Мартынов С. А.

Работу принял преподаватель _____ Душутина Е. В.

Оглавление

Постановка задачи	3
Исключения с помощью WinAPI	5
Использование GetExceptionCode	11
Пользовательская функция-фильтр	16
Использование RaiseException	22
Не обрабатываемые исключения	29
Вложенные исключения	36
Выход при помощи goto	40
Выход при помощи __leave	44
Преобразование SEH в C++ исключение	48
Финальный обработчик finally	53
Использование функции AbnormalTermination	57
Заключение	62

Постановка задачи

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 UnhandledExceptionFilter и SetUnhandledExceptionFilter 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 C, используя функцию `translator`;
10. Использовать финальный обработчик `finally`;
11. Проверить корректность выхода из блока `__try` с помощью функции `AbnormalTermination` в финальном обработчике `__finally`.

На каждый пункт представить отдельную программу, специфический код, связанный с особенностями генерации заданного исключения структурировать в отдельный элемент (функцию, макрос или иное).

В данной работе рассматриваются следующие исключения:

- **EXCEPTION_FLT_DIVIDE_BY_ZERO** - поток попытался сделать деление на ноль с плавающей точкой;
- **EXCEPTION_FLT_OVERFLOW** - переполнение при операции над числами с плавающей точкой.

Все результаты, представленные в данном отчёте получены с использованием Microsoft Windows 7 Ultimate Service Pack 1 64-bit (build 7601). Для разработки использовалась Microsoft Visual Studio Express 2013 for Windows Desktops (Version 12.0.30723.00 Update 3). В качестве отладчика использовался Microsoft WinDbg (release 6.3.9600.16384).

Исходный код всех представленных листингов доступен по адресу https://github.com/SemenMartynov/SPbPU_SystemProgramming.

Исключения с помощью WinAPI

Задачей этого раздела является генерирование и обработка исключений с помощью функций WinAPI.

В листинге 1 показана работа с исключениями. В зависимости от параметра, передаваемого при запуске, вызывается либо исключение деления на ноль, либо переполнение разрядной сетки при работе с типом float. Особо стоит обратить внимание на две вещи: изначально, все ошибки типа float маскируются, и для получения исключений нужно от этого маскирования избавиться (см. стр. 64-65); кроме того, операции с плавающими точками выполняются асинхронно, и нужно на этапе компиляции отключить расширения векторизации.

В 69-й строке используется квалификатор volatile, это помогает обмануть статический анализатор среды разработки (visual studio), который честно сигнализирует о явной ошибке (делении на ноль) и не позволяет собрать программу.

Листинг 1: Генерация и обработка исключения с помощью функций WinAPI

```
1  /* Task 1.
2     Generate and handle exceptions using the WinAPI functions;
3     */
4
5  // IMPORTANT: Don't forget to disable Enhanced Instructions!!!
6  // Properties -> Configuration Properties -> C/C++ -> Code Generation ->
7  // Enable Enhanced Instruction Set = No Enhanced Instructions (/arch:IA32)
8
9  #include <stdio.h>
10 #include <tchar.h>
11 #include <cstring>
12 #include <cfloat>
13 #include <cmath>
14 #include <excpt.h>
15 #include <windows.h>
16 #include <time.h>
17
18 // log
19 FILE* logfile;
20
```

```

21 void usage(const _TCHAR* prog);
22 void initlog(const _TCHAR* prog);
23 void closelog();
24 void writelog(_TCHAR* format, ...);
25
26 // Task switcher
27 enum {
28     DIVIDE_BY_ZERO,
29     FLT_OVERFLOW
30 } task;
31
32 // Defines the entry point for the console application.
33 int _tmain(int argc, _TCHAR* argv[]) {
34     //Init log
35     initlog(argv[0]);
36
37     // Check parameters number
38     if (argc != 2) {
39         _tprintf(_T("Too few parameters.\n\n"));
40         writelog(_T("Too few parameters."));
41         usage(argv[0]);
42         closelog();
43         exit(1);
44     }
45
46     // Set task
47     if (!_tcscmp(_T("-d"), argv[1])) {
48         task = DIVIDE_BY_ZERO;
49         writelog(_T("Task: DIVIDE_BY_ZERO exception."));
50     }
51     else if (!_tcscmp(_T("-o"), argv[1])) {
52         task = FLT_OVERFLOW;
53         writelog(_T("Task: FLT_OVERFLOW exception."));
54     }
55     else {
56         _tprintf(_T("Can't parse parameters.\n\n"));
57         writelog(_T("Can't parse parameters."));
58         usage(argv[0]);
59         closelog();
60         exit(1);
61     }
62
63     // Floating point exceptions are masked by default.
64     _clearfp();
65     _controlfp_s(NULL, 0, _EM_OVERFLOW | _EM_ZERODIVIDE);

```

```

66
67 // Set exception
68 __try {
69     volatile float tmp = 0;
70     switch (task) {
71     case DIVIDE_BY_ZERO:
72         writelog(_T("Ready for generate DIVIDE_BY_ZERO exception.));
73         tmp = 1 / tmp;
74         writelog(_T("DIVIDE_BY_ZERO exception is generated.));
75         break;
76     case FLT_OVERFLOW:
77         // Note: floating point execution happens asynchronously.
78         // So, the exception will not be handled until the next floating
79         // point instruction.
80         writelog(_T("Ready for generate FLT_OVERFLOW exception.));
81         tmp = pow(FLT_MAX, 3);
82         writelog(_T("Task: FLT_OVERFLOW exception is generated.));
83         break;
84     default:
85         break;
86     }
87 }
88 __except (EXCEPTION_EXECUTE_HANDLER) {
89     _tprintf(_T("Well, it looks like we caught something.));
90     writelog(_T("Exception is caught.));
91 }
92
93 closelog();
94 exit(0);
95 }
96
97 // Usage manual
98 void usage(const _TCHAR* prog) {
99     _tprintf(_T("Usage: \n"));
100     _tprintf(_T("\t%s -d\n"), prog);
101     _tprintf(_T("\t\t\t for exception float divide by zero,\n"));
102     _tprintf(_T("\t%s -o\n"), prog);
103     _tprintf(_T("\t\t\t for exception float overflow.\n"));
104 }
105
106 void initlog(const _TCHAR* prog) {
107     _TCHAR logname[255];
108     wcscpy_s(logname, prog);
109
110     // replace extension

```

```

111  _TCHAR* extension;
112  extension = wcsstr(logname, _T(".exe"));
113  wcsncpy_s(extension, 5, _T(".log"), 4);
114
115  // Try to open log file for append
116  if (_wfopen_s(&logfile, logname, _T("a+"))) {
117      _wprintf(_T("The following error occurred"));
118      _tprintf(_T("Can't open log file %s\n"), logname);
119      exit(1);
120  }
121
122  writelog(_T("%s is starting."), prog);
123 }
124
125 void closelog() {
126     writelog(_T("Shutting down.\n"));
127     fclose(logfile);
128 }
129
130 void writelog(_TCHAR* format, ...) {
131     _TCHAR buf[255];
132     va_list ap;
133
134     struct tm newtime;
135     __time64_t long_time;
136
137     // Get time as 64-bit integer.
138     _time64(&long_time);
139     // Convert to local time.
140     _localtime64_s(&newtime, &long_time);
141
142     // Convert to normal representation.
143     swprintf_s(buf, _T("[%d/%d/%d %d:%d:%d] "), newtime.tm_mday,
144         newtime.tm_mon + 1, newtime.tm_year + 1900, newtime.tm_hour,
145         newtime.tm_min, newtime.tm_sec);
146
147     // Write date and time
148     fwprintf(logfile, _T("%s"), buf);
149     // Write all params
150     va_start(ap, format);
151     _vsnwprintf_s(buf, sizeof(buf) - 1, format, ap);
152     fwprintf(logfile, _T("%s"), buf);
153     va_end(ap);
154     // New sting
155     fwprintf(logfile, _T("\n"));

```


Если запустить этот код в отладчике (рисунок 2), и пройти его по шагам, то можно увидеть, как управление с 73-й строки (где происходит исключительная ситуация), передаётся на 88-ю (в которой ожидается исключение), а потом на 93-ю (т.е. обратной передачи управления не происходит).

Произведём три запуска, первый раз без аргументом (для демонстрации зависимости исключения от передаваемого аргумента), второй раз с аргументом "d"(DIVIDE_BY_ZERO) и третий раз с аргументом "o"(FLT_OVERFLOW). Как видно на рисунке 1, первый запуск не дал результатов, второй и третий привёл к исключительной ситуации.

```

C:\Windows\system32\cmd.exe
Microsoft Windows [Version 6.1.7601]
Copyright (c) 2009 Microsoft Corporation. All rights reserved.

C:\Users\win7>"C:\Users\win7\Documents\Visual Studio 2013\Projects\ExceptionsProcessing\Debug\WinAPI.exe"
Too few parameters.

Usage:
C:\Users\win7\Documents\Visual Studio 2013\Projects\ExceptionsProcessing\Debug\WinAPI.exe -d
                                for exception float divide by zero,
C:\Users\win7\Documents\Visual Studio 2013\Projects\ExceptionsProcessing\Debug\WinAPI.exe -o
                                for exception float overflow.

C:\Users\win7>"C:\Users\win7\Documents\Visual Studio 2013\Projects\ExceptionsProcessing\Debug\WinAPI.exe" -d
Well, it looks like we caught something.
C:\Users\win7>"C:\Users\win7\Documents\Visual Studio 2013\Projects\ExceptionsProcessing\Debug\WinAPI.exe" -o
Well, it looks like we caught something.
C:\Users\win7>

```

Рис. 1: Запуск программы, генерирующей исключения средствами WinAPI.

Во время второго и третьего запуска, управление сразу после исключения передавалось на 88-ю строку, это можно видеть по листингу 2, содержащему лог работы программы. Запись из 74-й и 82-й строки в нём отсутствует, т.к. управление до этих строк не дошло.

Листинг 2: Генерация и обработка исключения с помощью функций WinAPI

```

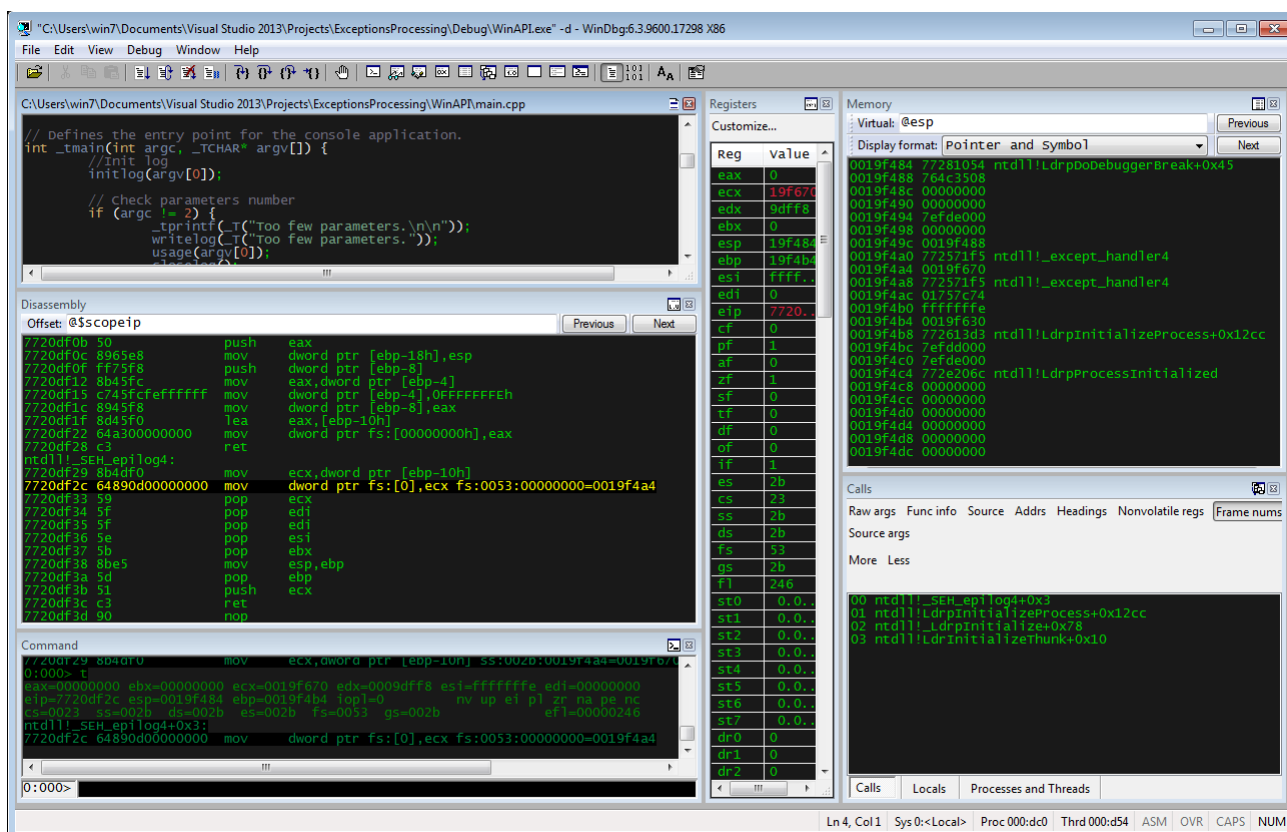
1 [6/2/2015 15:32:58] C:\Users\win7\Documents\Visual Studio 2013\Projects\
  ExceptionsProcessing\Debug\WinAPI.exe is starting.
2 [6/2/2015 15:32:58] Too few parameters.
3 [6/2/2015 15:32:58] Shutting down.
4
5 [6/2/2015 15:33:5] C:\Users\win7\Documents\Visual Studio 2013\Projects\
  ExceptionsProcessing\Debug\WinAPI.exe is starting.
6 [6/2/2015 15:33:5] Task: DIVIDE_BY_ZERO exception.
7 [6/2/2015 15:33:5] Ready for generate DIVIDE_BY_ZERO exception.

```

```

8 [6/2/2015 15:33:5] Exception is caught.
9 [6/2/2015 15:33:5] Shutting down.
10
11 [6/2/2015 15:33:10] C:\Users\win7\Documents\Visual Studio 2013\Projects\
    ExceptionsProcessing\Debug\WinAPI.exe is starting.
12 [6/2/2015 15:33:10] Task: FLT_OVERFLOW exception.
13 [6/2/2015 15:33:10] Ready for generate FLT_OVERFLOW exception.
14 [6/2/2015 15:33:10] Exception is caught.
15 [6/2/2015 15:33:10] Shutting down.

```



Благодаря тому, что исключение было обработано, оно не дошло до уровня операционной системы, и не было отражено в системном журнале.

Использование GetExceptionCode

Функция GetExceptionCode позволяет получить код исключения, которое было сгенерировано в процессе работы программы (смотри листинг 3). В первом случае (строка 79) она участвует в сравнении с макро-константой EXCEPTION_FLT_DIVIDE_BY_ZERO для определения подходящего обработчика для исключительного события. Во-втором случае (строка 98) она используется уже внутри обработчика, позволяя определить, что исключение вызвано переполнением при операции с типом float.

Листинг 3: Получение кода исключения с помощью функции GetExceptionCode

```
1  /* Task 2.
2     Get the exceptions code using the GetExceptionCode gunction:
3     - Use this function in the filter expression;
4     - Use this function in the handler.
5  */
6
7  // IMPORTANT: Don't forget to disable Enhanced Instructions!!!
8  // Properties -> Configuration Properties -> C/C++ -> Code Generation ->
9  // Enable Enhanced Instruction Set = No Enhanced Instructions (/arch:IA32)
10
11 #include <stdio.h>
12 #include <tchar.h>
13 #include <cstring>
14 #include <cfloat>
15 #include <cmath>
16 #include <excpt.h>
17 #include <windows.h>
18 #include <time.h>
19
20 // log
21 FILE* logfile;
22
23 void usage(const _TCHAR* prog);
24 void initlog(const _TCHAR* prog);
25 void closelog();
26 void writelog(_TCHAR* format, ...);
```

```

27
28 // Task switcher
29 enum {
30     DIVIDE_BY_ZERO,
31     FLT_OVERFLOW
32 } task;
33
34 // Defines the entry point for the console application.
35 int _tmain(int argc, _TCHAR* argv[]) {
36     //Init log
37     initlog(argv[0]);
38
39     // Check parameters number
40     if (argc != 2) {
41         _tprintf(_T("Too few parameters.\n\n"));
42         writelog(_T("Too few parameters. "));
43         usage(argv[0]);
44         closelog();
45         exit(1);
46     }
47
48     // Set task
49     if (!_tcscmp(_T("-d"), argv[1])) {
50         task = DIVIDE_BY_ZERO;
51         writelog(_T("Task: DIVIDE_BY_ZERO exception. "));
52     }
53     else if (!_tcscmp(_T("-o"), argv[1])) {
54         task = FLT_OVERFLOW;
55         writelog(_T("Task: FLT_OVERFLOW exception. "));
56     }
57     else {
58         _tprintf(_T("Can't parse parameters.\n\n"));
59         writelog(_T("Can't parse parameters. "));
60         usage(argv[0]);
61         closelog();
62         exit(1);
63     }
64
65     // Floating point exceptions are masked by default.
66     _clearfp();
67     _controlfp_s(NULL, 0, _EM_OVERFLOW | _EM_ZERODIVIDE);
68
69     // Set exception
70     volatile float tmp = 0;
71     switch (task) {

```

```

72 case DIVIDE_BY_ZERO:
73     __try {
74         writelog(_T("Ready for generate DIVIDE_BY_ZERO exception.));
75         tmp = 1 / tmp;
76         writelog(_T("DIVIDE_BY_ZERO exception is generated.));
77     }
78     // Use GetExceptionCode() function in the filter expression;
79     __except ((GetExceptionCode() == EXCEPTION_FLT_DIVIDE_BY_ZERO) ?
80 EXCEPTION_EXECUTE_HANDLER :
81             EXCEPTION_CONTINUE_SEARCH)
82     {
83         _tprintf(_T("Caught exception is: EXCEPTION_FLT_DIVIDE_BY_ZERO"));
84         writelog(_T("Caught exception is: EXCEPTION_FLT_DIVIDE_BY_ZERO"));
85     }
86     break;
87 case FLT_OVERFLOW:
88     __try {
89         // Note: floating point execution happens asynchronously.
90         // So, the exception will not be handled until the next
91         // floating point instruction.
92         writelog(_T("Ready for generate FLT_OVERFLOW exception.));
93         tmp = pow(FLT_MAX, 3);
94         writelog(_T("Task: FLT_OVERFLOW exception is generated.));
95     }
96     // Use GetExceptionCode() function in the handler.
97     __except (EXCEPTION_EXECUTE_HANDLER) {
98         if (GetExceptionCode() == EXCEPTION_FLT_OVERFLOW) {
99             writelog(_T("Caught exception is: EXCEPTION_FLT_OVERFLOW"));
100             _tprintf(_T("Caught exception is: EXCEPTION_FLT_OVERFLOW"));
101         }
102         else {
103             writelog(_T("UNKNOWN exception: %x\n"), GetExceptionCode());
104             _tprintf(_T("UNKNOWN exception: %x\n"), GetExceptionCode());
105         }
106     }
107     break;
108 default:
109     break;
110 }
111 closelog();
112 exit(0);
113 }
114
115 // Usage manual
116 void usage(const _TCHAR* prog) {

```

```

117     _tprintf(_T("Usage: \n"));
118     _tprintf(_T("\t%s -d\n"), prog);
119     _tprintf(_T("\t\t\t for exception float divide by zero.\n"));
120     _tprintf(_T("\t%s -o\n"), prog);
121     _tprintf(_T("\t\t\t for exception float overflow.\n"));
122 }
123
124 void initlog(const _TCHAR* prog) {
125     _TCHAR logname[255];
126     wcsncpy_s(logname, prog);
127
128     // replace extension
129     _TCHAR* extension;
130     extension = wcsstr(logname, _T(".exe"));
131     wcsncpy_s(extension, 5, _T(".log"), 4);
132
133     // Try to open log file for append
134     if (_wfopen_s(&logfile, logname, _T("a+"))) {
135         _wprintf(_T("The following error occurred"));
136         _tprintf(_T("Can't open log file %s\n"), logname);
137         exit(1);
138     }
139
140     writelog(_T("%s is starting."), prog);
141 }
142
143 void closelog() {
144     writelog(_T("Shutting down.\n"));
145     fclose(logfile);
146 }
147
148 void writelog(_TCHAR* format, ...) {
149     _TCHAR buf[255];
150     va_list ap;
151
152     struct tm newtime;
153     __time64_t long_time;
154
155     // Get time as 64-bit integer.
156     _time64(&long_time);
157     // Convert to local time.
158     _localtime64_s(&newtime, &long_time);
159
160     // Convert to normal representation.
161     swprintf_s(buf, _T("[%d/%d/%d %d:%d:%d] "), newtime.tm_mday,

```

```

162     newtime.tm_mon + 1, newtime.tm_year + 1900, newtime.tm_hour,
163     newtime.tm_min, newtime.tm_sec);
164
165     // Write date and time
166     fprintf(logfile, _T("%s"), buf);
167     // Write all params
168     va_start(ap, format);
169     _vsnwprintf_s(buf, sizeof(buf) - 1, format, ap);
170     fprintf(logfile, _T("%s"), buf);
171     va_end(ap);
172     // New sting
173     fprintf(logfile, _T("\n"));
174 }

```

Таким образом, рассмотрены два способа фильтрации исключений - на уровне входа в блок `__except`, либо уже непосредственно в обработчике (тогда в `__except` ставится макро-константа `EXCEPTION_EXECUTE_HANDLER`, позволяющая принимать любые исключения). Далее будет рассмотрен более логичный способ фильтрации исключений специальной функцией.

Запуск под отладчиком показывает картину практически аналогичную предыдущему случаю, но есть разница в логе работы программы (листинг 4). На этот раз мы знаем какое исключение произошло и это фиксируем в логе.

Листинг 4: Генерация и обработка исключения с помощью функций WinAPI

```

1 [6/2/2015 16:42:36] C:\Users\win7\Documents\Visual Studio 2013\Projects\
    ExceptionsProcessing\Debug\GetExceptionCode.exe is starting.
2 [6/2/2015 16:42:36] Task: DIVIDE_BY_ZERO exception.
3 [6/2/2015 16:42:36] Ready for generate DIVIDE_BY_ZERO exception.
4 [6/2/2015 16:42:36] Caught exception is: EXCEPTION_FLT_DIVIDE_BY_ZERO
5 [6/2/2015 16:42:36] Shutting down.
6
7 [6/2/2015 16:42:39] C:\Users\win7\Documents\Visual Studio 2013\Projects\
    ExceptionsProcessing\Debug\GetExceptionCode.exe is starting.
8 [6/2/2015 16:42:39] Task: FLT_OVERFLOW exception.
9 [6/2/2015 16:42:39] Ready for generate FLT_OVERFLOW exception.
10 [6/2/2015 16:42:39] Caught exception is: EXCEPTION_FLT_OVERFLOW
11 [6/2/2015 16:42:39] Shutting down.

```

Пользовательская функция-фильтр

В листинге 5 представлена функция-фильтр, которая возвращает EXCEPTION_CONTINUE_SEARCH только если исключение вызвано EXCEPTION_FLT_DIVIDE_BY_ZERO или EXCEPTION_FLT_OVERFLOW.

Листинг 5: Использование собственной функции фильтра

```
1  /* Task 3.
2     Create your own filter function.
3     */
4
5  // IMPORTANT: Don't forget to disable Enhanced Instructions!!!
6  // Properties -> Configuration Properties -> C/C++ -> Code Generation ->
7  // Enable Enhanced Instruction Set = No Enhanced Instructions (/arch:IA32)
8
9  #include <stdio.h>
10 #include <tchar.h>
11 #include <cstring>
12 #include <cfloat>
13 #include <cmath>
14 #include <excpt.h>
15 #include <windows.h>
16 #include <time.h>
17
18 // log
19 FILE* logfile;
20
21 void usage(const _TCHAR* prog);
22 void initlog(const _TCHAR* prog);
23 void closelog();
24 void writelog(_TCHAR* format, ...);
25 LONG Filter(DWORD dwExceptionCode);
26
27 // Task switcher
28 enum {
29     DIVIDE_BY_ZERO,
30     FLT_OVERFLOW
```



```

31 } task;
32
33 // Defines the entry point for the console application.
34 int _tmain(int argc, _TCHAR* argv[]) {
35     //Init log
36     initlog(argv[0]);
37
38     // Check parameters number
39     if (argc != 2) {
40         _tprintf(_T("Too few parameters.\n\n"));
41         writelog(_T("Too few parameters."));
42         usage(argv[0]);
43         closelog();
44         exit(1);
45     }
46
47     // Set task
48     if (!_tcscmp(_T("-d"), argv[1])) {
49         task = DIVIDE_BY_ZERO;
50         writelog(_T("Task: DIVIDE_BY_ZERO exception."));
51     }
52     else if (!_tcscmp(_T("-o"), argv[1])) {
53         task = FLT_OVERFLOW;
54         writelog(_T("Task: FLT_OVERFLOW exception."));
55     }
56     else {
57         _tprintf(_T("Can't parse parameters.\n\n"));
58         writelog(_T("Can't parse parameters."));
59         usage(argv[0]);
60         closelog();
61         exit(1);
62     }
63
64     // Floating point exceptions are masked by default.
65     _clearfp();
66     _controlfp_s(NULL, 0, _EM_OVERFLOW | _EM_ZERODIVIDE);
67
68     // Set exception
69     __try {
70         volatile float tmp = 0;
71         switch (task) {
72             case DIVIDE_BY_ZERO:
73                 writelog(_T("Ready for generate DIVIDE_BY_ZERO exception."));
74                 tmp = 1 / tmp;
75                 writelog(_T("DIVIDE_BY_ZERO exception is generated."));

```

```

76     break;
77 case FLT_OVERFLOW:
78     // Note: floating point execution happens asynchronously.
79     // So, the exception will not be handled until the next floating
80     // point instruction.
81     writelog(_T("Ready for generate FLT_OVERFLOW exception.));
82     tmp = pow(FLT_MAX, 3);
83     writelog(_T("Task: FLT_OVERFLOW exception is generated.));
84     break;
85 default:
86     break;
87 }
88 }
89 // Own filter function.
90 __except (Filter(GetExceptionCode())) {
91     printf("Caught exception is: ");
92     switch (GetExceptionCode()){
93     case EXCEPTION_FLT_DIVIDE_BY_ZERO:
94         _tprintf(_T("Caught exception is: EXCEPTION_FLT_DIVIDE_BY_ZERO"));
95         writelog(_T("Caught exception is: EXCEPTION_FLT_DIVIDE_BY_ZERO"));
96         break;
97     case EXCEPTION_FLT_OVERFLOW:
98         _tprintf(_T("Caught exception is: EXCEPTION_FLT_OVERFLOW"));
99         writelog(_T("Caught exception is: EXCEPTION_FLT_OVERFLOW"));
100        break;
101    default:
102        _tprintf(_T("UNKNOWN exception: %x\n"), GetExceptionCode());
103        writelog(_T("UNKNOWN exception: %x\n"), GetExceptionCode());
104    }
105 }
106 closelog();
107 exit(0);
108 }
109
110 // Own filter function.
111 LONG Filter(DWORD dwExceptionCode) {
112     _tprintf(_T("Filter function used"));
113     if (dwExceptionCode == EXCEPTION_FLT_DIVIDE_BY_ZERO || dwExceptionCode ==
        EXCEPTION_FLT_OVERFLOW)
114         return EXCEPTION_EXECUTE_HANDLER;
115     return EXCEPTION_CONTINUE_SEARCH;
116 }
117
118 // Usage manual
119 void usage(const _TCHAR* prog) {

```

```

120     _tprintf(_T("Usage: \n"));
121     _tprintf(_T("\t%s -d\n"), prog);
122     _tprintf(_T("\t\t\t for exception float divide by zero.\n"));
123     _tprintf(_T("\t%s -o\n"), prog);
124     _tprintf(_T("\t\t\t for exception float overflow.\n"));
125 }
126
127 void initlog(const _TCHAR* prog) {
128     _TCHAR logname[255];
129     wcsncpy_s(logname, prog);
130
131     // replace extension
132     _TCHAR* extension;
133     extension = wcsstr(logname, _T(".exe"));
134     wcsncpy_s(extension, 5, _T(".log"), 4);
135
136     // Try to open log file for append
137     if (_wfopen_s(&logfile, logname, _T("a+"))) {
138         _wprintf(_T("The following error occurred"));
139         _tprintf(_T("Can't open log file %s\n"), logname);
140         exit(1);
141     }
142
143     writelog(_T("%s is starting."), prog);
144 }
145
146 void closelog() {
147     writelog(_T("Shutting down.\n"));
148     fclose(logfile);
149 }
150
151 void writelog(_TCHAR* format, ...) {
152     _TCHAR buf[255];
153     va_list ap;
154
155     struct tm newtime;
156     __time64_t long_time;
157
158     // Get time as 64-bit integer.
159     _time64(&long_time);
160     // Convert to local time.
161     _localtime64_s(&newtime, &long_time);
162
163     // Convert to normal representation.
164     swprintf_s(buf, _T("[%d/%d/%d %d:%d:%d] "), newtime.tm_mday,

```

```

165     newtime.tm_mon + 1, newtime.tm_year + 1900, newtime.tm_hour,
166     newtime.tm_min, newtime.tm_sec);
167
168     // Write date and time
169     fprintf(logfile, _T("%s"), buf);
170     // Write all params
171     va_start(ap, format);
172     _vsnwprintf_s(buf, sizeof(buf) - 1, format, ap);
173     fprintf(logfile, _T("%s"), buf);
174     va_end(ap);
175     // New sting
176     fprintf(logfile, _T("\n"));
177 }

```

При изучении работы программы под отладчиком, выяснилось что при возникновении исключения, управление не передаётся в то место, где определена функция-фильтр. Вероятно компилятор оптимизирует код и подставляет её целиком на место вызова. На рисунке 3 видна передача управления на обработку исключения после работы функции-фильтра.

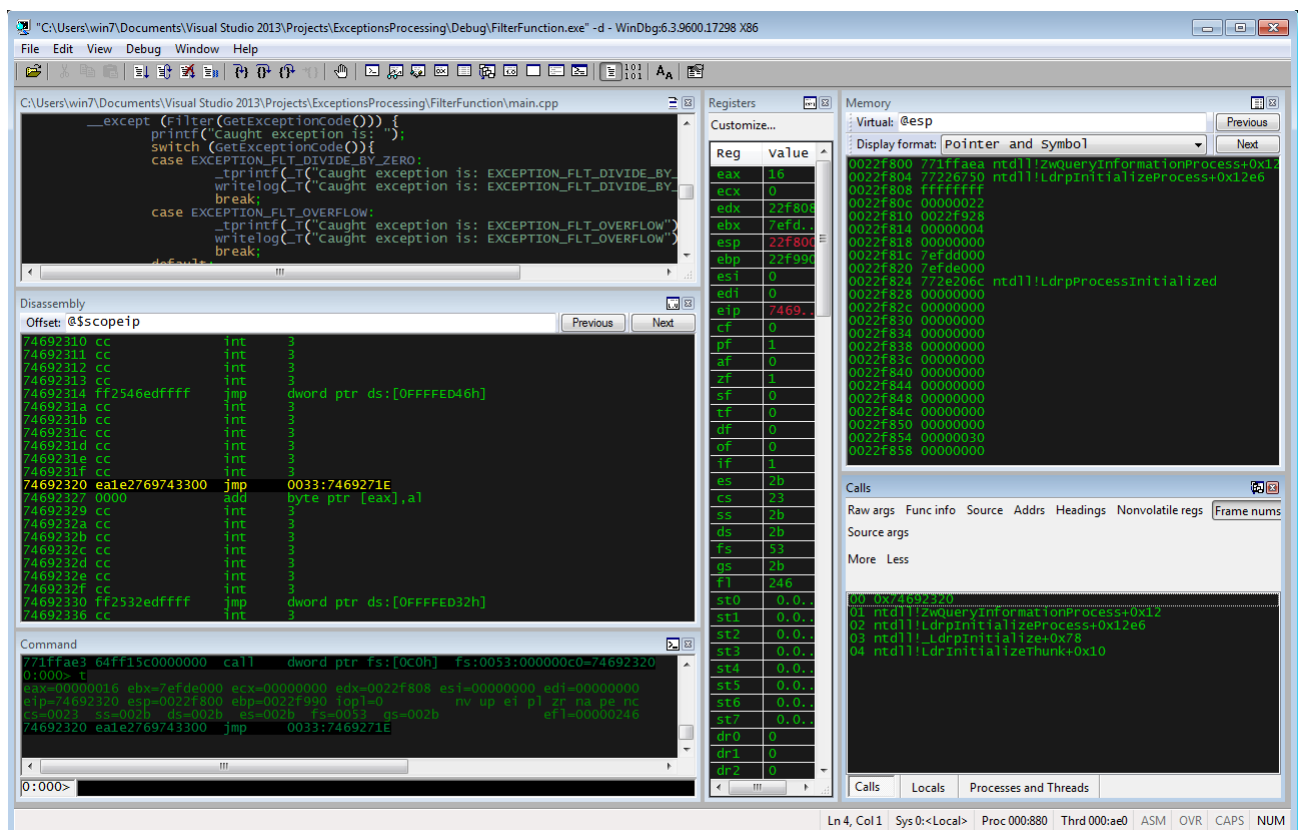


Рис. 3: Передача управления обработчику, после отработки функции-фильтра

В обработчике фактически происходит только вызов функций логирования (листинг 6). Как и раньше, строки 75 и 83 оказываются пропущенными, т.к. после возбуждения исключения управление переходит обработчику, нарушая линейный порядок.

Листинг 6: Генерация и обработка исключения с помощью функций WinAPI

```
1 [6/2/2015 16:45:29] C:\Users\win7\Documents\Visual Studio 2013\Projects\  
    ExceptionsProcessing\Debug\FilterFunction.exe is starting.  
2 [6/2/2015 16:45:29] Task: DIVIDE_BY_ZERO exception.  
3 [6/2/2015 16:45:29] Ready for generate DIVIDE_BY_ZERO exception.  
4 [6/2/2015 16:45:29] Caught exception is: EXCEPTION_FLT_DIVIDE_BY_ZERO  
5 [6/2/2015 16:45:29] Shutting down.  
6  
7 [6/2/2015 16:45:34] C:\Users\win7\Documents\Visual Studio 2013\Projects\  
    ExceptionsProcessing\Debug\FilterFunction.exe is starting.  
8 [6/2/2015 16:45:34] Task: FLT_OVERFLOW exception.  
9 [6/2/2015 16:45:34] Ready for generate FLT_OVERFLOW exception.  
10 [6/2/2015 16:45:34] Caught exception is: EXCEPTION_FLT_OVERFLOW  
11 [6/2/2015 16:45:34] Shutting down.
```

Использование RaiseException

Исключение можно возбудить не только в результате каких-то арифметических или логических операций, но и искусственным образом, вызвав функцию RaiseException. Она обладает 4-я параметрами, но наиболее важным является первый, который определяет тип возбуждаемого исключения. Работа этой функции показана в листинге 7.

Листинг 7: Программная генерация исключения при помощи функции RaiseException

```
1  /* Task 4.
2     Get information about the exception using the GetExceptionInformation()
3     fnc;
4     throw an exception using the RaiseException() function.
5     */
6
7  // IMPORTANT: Don't forget to disable Enhanced Instructions!!!
8  // Properties -> Configuration Properties -> C/C++ -> Code Generation ->
9  // Enable Enhanced Instruction Set = No Enhanced Instructions (/arch:IA32)
10
11 #include <stdio.h>
12 #include <tchar.h>
13 #include <cstring>
14 #include <cfloat>
15 #include <cmath>
16 #include <excpt.h>
17 #include <windows.h>
18 #include <time.h>
19
20 // log
21 FILE* logfile;
22
23 void usage(const _TCHAR* prog);
24 void initlog(const _TCHAR* prog);
25 void closelog();
26 void writelog(_TCHAR* format, ...);
27
28 LONG Filter(DWORD dwExceptionCode, const _EXCEPTION_POINTERS *ep);
29
30 // Task switcher
```

```

29 enum {
30     DIVIDE_BY_ZERO,
31     FLT_OVERFLOW
32 } task;
33
34 // Defines the entry point for the console application.
35 int _tmain(int argc, _TCHAR* argv[]) {
36     //Init log
37     initlog(argv[0]);
38
39     // Check parameters number
40     if (argc != 2) {
41         _tprintf(_T("Too few parameters.\n\n"));
42         writelog(_T("Too few parameters."));
43         usage(argv[0]);
44         closelog();
45         exit(1);
46     }
47
48     // Set task
49     if (!_tcscmp(_T("-d"), argv[1])) {
50         task = DIVIDE_BY_ZERO;
51         writelog(_T("Task: DIVIDE_BY_ZERO exception."));
52     }
53     else if (!_tcscmp(_T("-o"), argv[1])) {
54         task = FLT_OVERFLOW;
55         writelog(_T("Task: FLT_OVERFLOW exception."));
56     }
57     else {
58         _tprintf(_T("Can't parse parameters.\n\n"));
59         writelog(_T("Can't parse parameters."));
60         usage(argv[0]);
61         closelog();
62         exit(1);
63     }
64
65     // Floating point exceptions are masked by default.
66     _clearfp();
67     _controlfp_s(NULL, 0, _EM_OVERFLOW | _EM_ZERODIVIDE);
68
69     __try {
70         switch (task) {
71             case DIVIDE_BY_ZERO:
72                 // throw an exception using the RaiseException() function
73                 writelog(_T("Ready for generate DIVIDE_BY_ZERO exception.));

```

```

74     RaiseException(EXCEPTION_FLT_DIVIDE_BY_ZERO,
75         EXCEPTION_NONCONTINUABLE, 0, NULL);
76     writelog(_T("DIVIDE_BY_ZERO exception is generated.));
77     break;
78 case FLT_OVERFLOW:
79     // throw an exception using the RaiseException() function
80     writelog(_T("Ready for generate FLT_OVERFLOW exception.));
81     RaiseException(EXCEPTION_FLT_OVERFLOW,
82         EXCEPTION_NONCONTINUABLE, 0, NULL);
83     writelog(_T("Task: FLT_OVERFLOW exception is generated.));
84     break;
85 default:
86     break;
87 }
88 }
89 __except (Filter(GetExceptionCode(), GetExceptionInformation())) {
90     // There is nothing to do, everything is done in the filter function.
91 }
92 closelog();
93 exit(0);
94 }
95
96 LONG Filter(DWORD dwExceptionCode, const _EXCEPTION_POINTERS *
    ExceptionPointers) {
97     enum { size = 200 };
98     _TCHAR buf[size] = { '\0' };
99     const _TCHAR* err = _T("Fatal error!\nexception code: 0x");
100    const _TCHAR* mes = _T("\nProgram terminate!");
101    if (ExceptionPointers)
102        // Get information about the exception using the GetExceptionInformation
103        swprintf_s(buf, _T("%s%x%s%x%s%x"), err,
104            ExceptionPointers->ExceptionRecord->ExceptionCode,
105            _T(", data address: 0x"),
106            ExceptionPointers->ExceptionRecord->ExceptionInformation[1],
107            _T(", instruction address: 0x"),
108            ExceptionPointers->ExceptionRecord->ExceptionAddress, mes);
109    else
110        swprintf_s(buf, _T("%s%x%s"), err, dwExceptionCode, mes);
111
112    _tprintf(_T("%s"), buf);
113    writelog(_T("%s"), buf);
114
115    return EXCEPTION_EXECUTE_HANDLER;
116 }
117

```



```

118 // Usage manual
119 void usage(const _TCHAR* prog) {
120     _tprintf(_T("Usage: \n"));
121     _tprintf(_T("\t%s -d\n"), prog);
122     _tprintf(_T("\t\t\t for exception float divide by zero.\n"));
123     _tprintf(_T("\t%s -o\n"), prog);
124     _tprintf(_T("\t\t\t for exception float overflow.\n"));
125 }
126
127 void initlog(const _TCHAR* prog) {
128     _TCHAR logname[255];
129     wcsncpy_s(logname, prog);
130
131     // replace extension
132     _TCHAR* extension;
133     extension = wcsstr(logname, _T(".exe"));
134     wcsncpy_s(extension, 5, _T(".log"), 4);
135
136     // Try to open log file for append
137     if (_wfopen_s(&logfile, logname, _T("a+"))) {
138         _wprintf(_T("The following error occurred"));
139         _tprintf(_T("Can't open log file %s\n"), logname);
140         exit(1);
141     }
142
143     writelog(_T("%s is starting."), prog);
144 }
145
146 void closelog() {
147     writelog(_T("Shutting down.\n"));
148     fclose(logfile);
149 }
150
151 void writelog(_TCHAR* format, ...) {
152     _TCHAR buf[255];
153     va_list ap;
154
155     struct tm newtime;
156     __time64_t long_time;
157
158     // Get time as 64-bit integer.
159     _time64(&long_time);
160     // Convert to local time.
161     _localtime64_s(&newtime, &long_time);
162

```

```

163 // Convert to normal representation.
164 swprintf_s(buf, _T("[%d/%d/%d %d:%d:%d] "), newtime.tm_mday,
165         newtime.tm_mon + 1, newtime.tm_year + 1900, newtime.tm_hour,
166         newtime.tm_min, newtime.tm_sec);
167
168 // Write date and time
169 fwprintf(logfile, _T("%s"), buf);
170 // Write all params
171 va_start(ap, format);
172 _vsnwprintf_s(buf, sizeof(buf) - 1, format, ap);
173 fwprintf(logfile, _T("%s"), buf);
174 va_end(ap);
175 // New sting
176 fwprintf(logfile, _T("\n"));
177 }

```

Информацию об исключении можно получить из функции `GetExceptionInformation`, которая, в действительности, никакой информацией не владеет но возвращает указатель на структуру `EXCEPTION_POINTERS`. В свою очередь, эта структура содержит два указателя на `ExceptionRecord` и на `ContextRecord`, в которых уже находится информация об исключении.

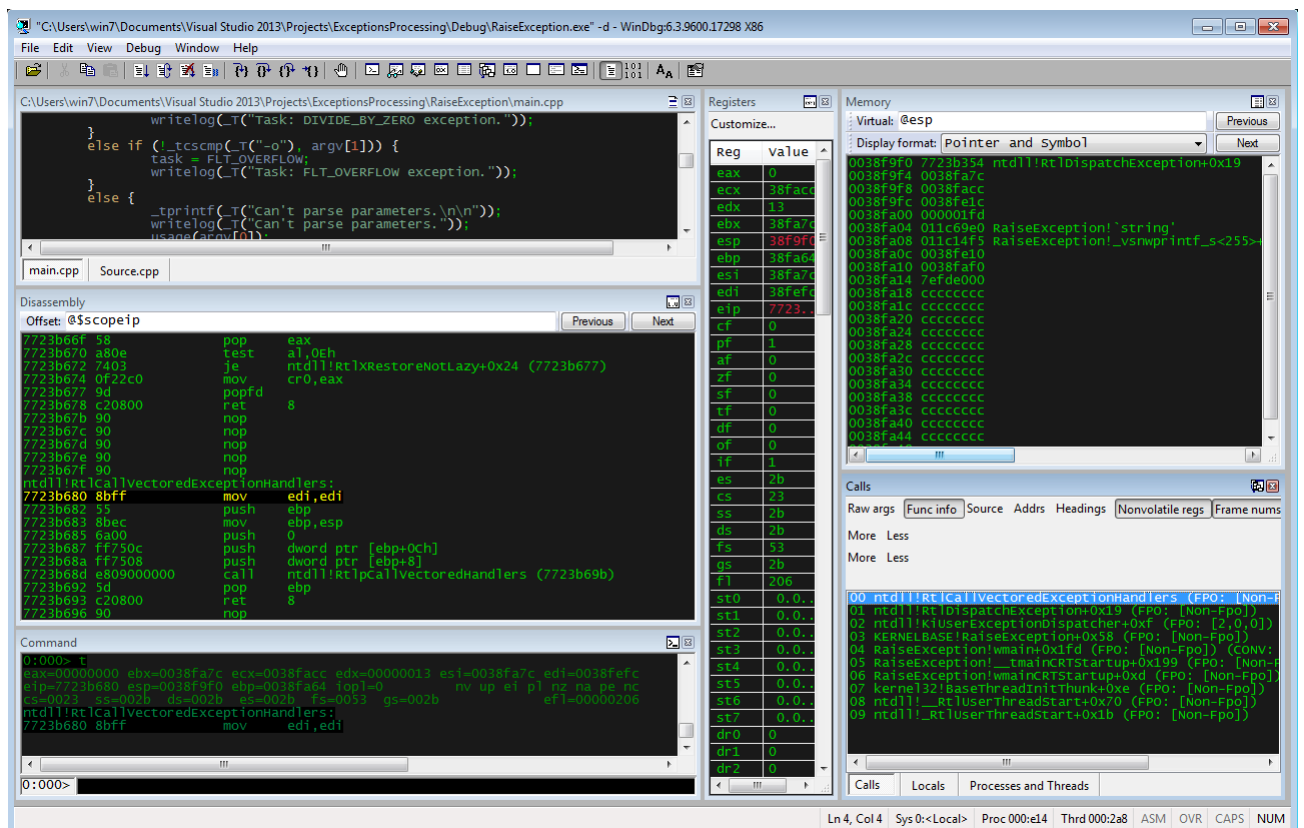


Рис. 4: Информация об исключении доступна в процессе работы функции-фильтра

Важной особенностью функции `GetExceptionInformation` является то, что ее можно вызывать только в функции-фильтре исключений, т.к. структуры `CONTEXT`, `EXCEPTION_RECORD` и `EXCEPTION_POINTERS` существуют лишь во время обработки фильтра исключения. В момент, когда управление переходит к обработчику исключений, эти данные в стеке разрушаются. На рисунке 4 показан момент получения информации о возникшем исключении. Обработчик исключения находится выше по стеку, и когда ему будет возвращено управление от функции фильтра стек уже будет зачищен.

```

C:\Windows\system32\cmd.exe
C:\Users\win7>"C:\Users\win7\Documents\Visual Studio 2013\Projects\ExceptionsProcessing\Debug\RaiseException.exe" -o
Fatal error!
exception code: 0xc0000091, data address: 0xfefefefe, instruction address: 0x75abc42d
C:\Users\win7>"C:\Users\win7\Documents\Visual Studio 2013\Projects\ExceptionsProcessing\Debug\RaiseException.exe" -d
Fatal error!
exception code: 0xc000008e, data address: 0xfefefefe, instruction address: 0x75abc42d
C:\Users\win7>
C:\Users\win7>
C:\Users\win7>
C:\Users\win7>
C:\Users\win7>
C:\Users\win7>
C:\Users\win7>
C:\Users\win7>
C:\Users\win7>
C:\Users\win7>
C:\Users\win7>
C:\Users\win7>
C:\Users\win7>
C:\Users\win7>
C:\Users\win7>

```

Рис. 5: Передача управления обработчику, после отработки функции-фильтра

На рисунке 5 показан вызов программы, генерирующей исключения программным образом, а в листинге 8 представлен лог её работы.

Листинг 8: Программная генерация исключения и получение информации о нём

```

1 [6/2/2015 17:7:22] C:\Users\win7\Documents\Visual Studio 2013\Projects\
  ExceptionsProcessing\Debug\RaiseException.exe is starting.
2 [6/2/2015 17:7:22] Task: FLT_OVERFLOW exception.
3 [6/2/2015 17:7:22] Ready for generate FLT_OVERFLOW exception.
4 [6/2/2015 17:7:22] Fatal error!
5 exception code: 0xc0000091, data address: 0xfefefefe, instruction address: 0
  x75abc42d
6 [6/2/2015 17:7:22] Shutting down.
7
8 [6/2/2015 17:7:26] C:\Users\win7\Documents\Visual Studio 2013\Projects\
  ExceptionsProcessing\Debug\RaiseException.exe is starting.
9 [6/2/2015 17:7:26] Task: DIVIDE_BY_ZERO exception.
10 [6/2/2015 17:7:26] Ready for generate DIVIDE_BY_ZERO exception.

```

```
11 [6/2/2015 17:7:26] Fatal error!  
12 exeption code: 0xc000008e, data adress: 0xfefefefe, instruction adress: 0  
    x75abc42d  
13 [6/2/2015 17:7:26] Shutting down.
```

Не обрабатываемые исключения

Если ни один из установленных программистом обработчиков не подошла для обработки исключения (либо программист вообще не установил ни один обработчик), то вызывается функция `UnhandledExceptionFilter`, которая выполняет проверку, запущен ли процесс под отладчиком, и информирует процесс, если отладчик доступен. Далее, функция вызывает фильтр умалчиваемого обработчика (который устанавливается функцией `SetUnhandledExceptionFilter` и который возвращает `EXCEPTION_EXECUTE_HANDLER`). Затем, в зависимости от настроек операционной системы, вызывается либо отладчик, либо функция `NtRaiseHardError`, которая отображает сообщение об ошибке.

Листинг 9 показывает работу с `UnhandledExceptionFilter`. Возвращаемое значение определяется в строках 108 и 109.

Листинг 9: Необработанные исключения

```
1  /* Task 5.
2     Use the UnhandledExceptionFilter and SetUnhandledExceptionFilter
3     for unhandled exceptions;.
4     */
5
6  // IMPORTANT: Don't forget to disable Enhanced Instructions!!!
7  // Properties -> Configuration Properties -> C/C++ -> Code Generation ->
8  // Enable Enhanced Instruction Set = No Enhanced Instructions (/arch:IA32)
9
10 #include <stdio.h>
11 #include <tchar.h>
12 #include <cstring>
13 #include <cfloat>
14 #include <cmath>
15 #include <excpt.h>
16 #include <windows.h>
17 #include <time.h>
18
19 // log
20 FILE* logfile;
21
22 void usage(const _TCHAR* prog);
```

```

23 void initlog(const _TCHAR* prog);
24 void closelog();
25 void writelog(_TCHAR* format, ...);
26
27 // prototype
28 LONG WINAPI MyUnhandledExceptionFilter(EXCEPTION_POINTERS* ExceptionInfo);
29
30 // Task switcher
31 enum {
32     DIVIDE_BY_ZERO,
33     FLT_OVERFLOW
34 } task;
35
36 // Defines the entry point for the console application.
37 int _tmain(int argc, _TCHAR* argv[]) {
38     //Init log
39     initlog(argv[0]);
40
41     // Check parameters number
42     if (argc != 2) {
43         _tprintf(_T("Too few parameters.\n\n"));
44         writelog(_T("Too few parameters."));
45         usage(argv[0]);
46         closelog();
47         exit(1);
48     }
49
50     // Set task
51     if (!_tcscmp(_T("-d"), argv[1])) {
52         task = DIVIDE_BY_ZERO;
53         writelog(_T("Task: DIVIDE_BY_ZERO exception."));
54     }
55     else if (!_tcscmp(_T("-o"), argv[1])) {
56         task = FLT_OVERFLOW;
57         writelog(_T("Task: FLT_OVERFLOW exception."));
58     }
59     else {
60         _tprintf(_T("Can't parse parameters.\n\n"));
61         writelog(_T("Can't parse parameters."));
62         usage(argv[0]);
63         closelog();
64         exit(1);
65     }
66
67     // Floating point exceptions are masked by default.

```

```

68 _clearfp();
69 _controlfp_s(NULL, 0, _EM_OVERFLOW | _EM_ZERODIVIDE);
70
71 volatile float tmp = 0;
72 ::SetUnhandledExceptionFilter(MyUnhandledExceptionFilter);
73
74 switch (task) {
75 case DIVIDE_BY_ZERO:
76     // throw an exception using the RaiseException() function
77     writelog(_T("Ready for generate DIVIDE_BY_ZERO exception.));
78     RaiseException(EXCEPTION_FLT_DIVIDE_BY_ZERO,
79         EXCEPTION_EXECUTE_FAULT, 0, NULL);
80     writelog(_T("DIVIDE_BY_ZERO exception is generated.));
81     break;
82 case FLT_OVERFLOW:
83     // throw an exception using the RaiseException() function
84     writelog(_T("Ready for generate FLT_OVERFLOW exception.));
85     RaiseException(EXCEPTION_FLT_OVERFLOW,
86         EXCEPTION_EXECUTE_FAULT, 0, NULL);
87     writelog(_T("Task: FLT_OVERFLOW exception is generated.));
88     break;
89 default:
90     break;
91 }
92
93 closelog();
94 exit(0);
95 }
96
97 LONG WINAPI MyUnhandledExceptionFilter(EXCEPTION_POINTERS* ExceptionInfo) {
98     enum { size = 200 };
99     _TCHAR buf[size] = { '\0' };
100     const _TCHAR* err = _T("Unhandled exception!\nexception code : 0x");
101     // Get information about the exception using the GetExceptionInformation
102     swprintf_s(buf, _T("%s%x%s%x%s%x"), err, ExceptionInfo->ExceptionRecord->
        ExceptionCode,
103         _T(", data address: 0x"), ExceptionInfo->ExceptionRecord->
        ExceptionInformation[1],
104         _T(", instruction address: 0x"), ExceptionInfo->ExceptionRecord->
        ExceptionAddress);
105     _tprintf(_T("%s"), buf);
106     writelog(_T("%s"), buf);
107
108     return EXCEPTION_CONTINUE_SEARCH;
109     //return EXCEPTION_EXECUTE_HANDLER;

```

```

110 }
111
112 // Usage manual
113 void usage(const _TCHAR* prog) {
114     _tprintf(_T("Usage: \n"));
115     _tprintf(_T("\t%s -d\n"), prog);
116     _tprintf(_T("\t\t\t for exception float divide by zero,\n"));
117     _tprintf(_T("\t%s -o\n"), prog);
118     _tprintf(_T("\t\t\t for exception float overflow.\n"));
119 }
120
121 void initlog(const _TCHAR* prog) {
122     _TCHAR logname[255];
123     wcsncpy_s(logname, prog);
124
125     // replace extension
126     _TCHAR* extension;
127     extension = wcsstr(logname, _T(".exe"));
128     wcsncpy_s(extension, 5, _T(".log"), 4);
129
130     // Try to open log file for append
131     if (_wfopen_s(&logfile, logname, _T("a+"))) {
132         _wprintf(_T("The following error occurred"));
133         _tprintf(_T("Can't open log file %s\n"), logname);
134         exit(1);
135     }
136
137     writelog(_T("%s is starting."), prog);
138 }
139
140 void closelog() {
141     writelog(_T("Shutting down.\n"));
142     fclose(logfile);
143 }
144
145 void writelog(_TCHAR* format, ...) {
146     _TCHAR buf[255];
147     va_list ap;
148
149     struct tm newtime;
150     __time64_t long_time;
151
152     // Get time as 64-bit integer.
153     _time64(&long_time);
154     // Convert to local time.

```



```

155 _localtime64_s(&newtime, &long_time);
156
157 // Convert to normal representation.
158 swprintf_s(buf, _T("[%d/%d/%d %d:%d:%d] "), newtime.tm_mday,
159         newtime.tm_mon + 1, newtime.tm_year + 1900, newtime.tm_hour,
160         newtime.tm_min, newtime.tm_sec);
161
162 // Write date and time
163 fwprintf(logfile, _T("%s"), buf);
164 // Write all params
165 va_start(ap, format);
166 _vsnwprintf_s(buf, sizeof(buf) - 1, format, ap);
167 fwprintf(logfile, _T("%s"), buf);
168 va_end(ap);
169 // New sting
170 fwprintf(logfile, _T("\n"));
171 fflush(logfile);
172 }

```

Для начала запустим программу так, чтобы фильтр возвращал EXCEPTION_EXECUTE_HANDLE. Это считается нормальной ситуацией, и на рисунке 6 видно как происходит передача управления.

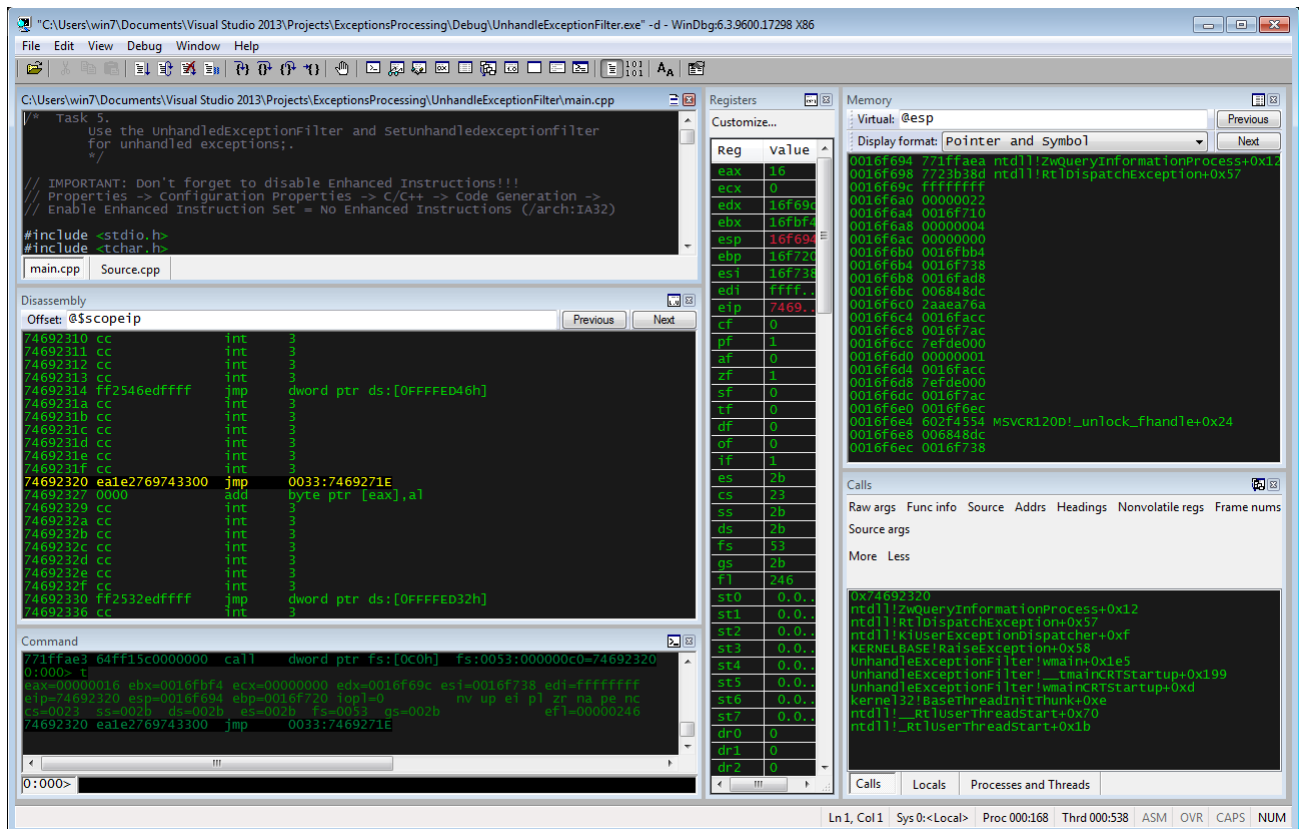


Рис. 6: Нормальная обработка исключения через фильтр

Запустим программу ещё раз, фильтр вернёт EXCEPTION_CONTINUE_SEARCH. Это событие будет передано операционной системе, и будет зафиксировано в системном журнале (рисунок 7). Что примечательно, обработчик успел выполнить свою задачу, информация об ошибке выведена на экран (рисунок 8) и сохранена в лог (листинг 10), системная ошибка возникла уже после.

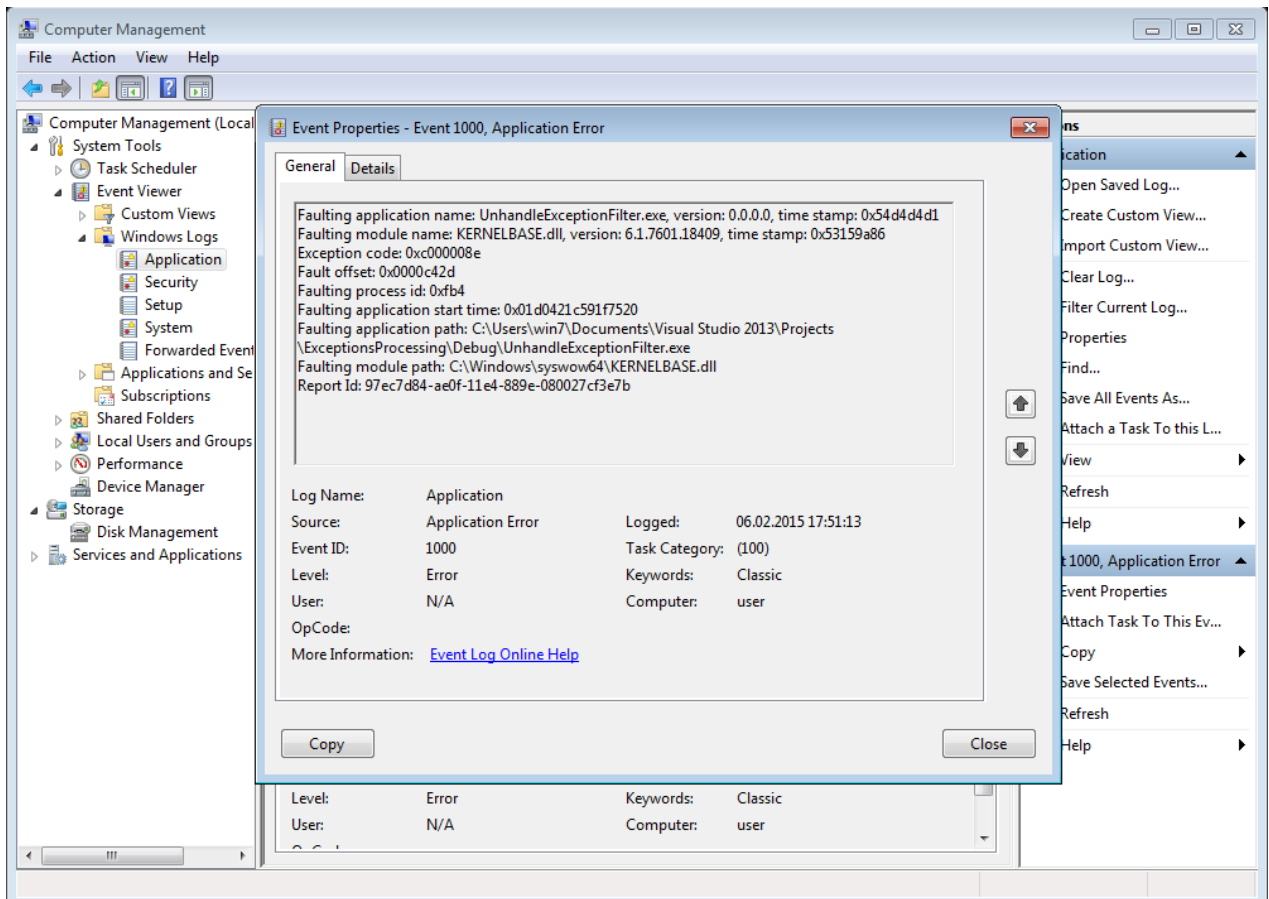


Рис. 7: Исключение зафиксировано в системном журнале

В логе программы можно прочитать информацию о произошедшей исключительной ситуации. Вместе с тем, можно видеть, что программа не была завершена корректно, а дескриптер файла-лога не был закрыт.

Листинг 10: Обработчик успел сохранить данные об исключении

```

1 [6/2/2015 18:9:43] C:\Users\win7\Documents\Visual Studio 2013\Projects\
  ExceptionsProcessing\Debug\UnhandleExceptionFilter.exe is starting.
2 [6/2/2015 18:9:43] Task: DIVIDE_BY_ZERO exception.
3 [6/2/2015 18:9:43] Ready for generate DIVIDE_BY_ZERO exception.
4 [6/2/2015 18:9:43] Unhandled exception!
5 exception code : 0xc000008e, data adress: 0xfefefefe, instruction adress: 0
  x75abc42d

```


Вложенные исключения

Листинг 11 показывает, как происходит передача исключения, в поисках подходящего обработчика. Самым ближайшим (по стеку) обработчиком для исключения, вызванного делением на 0, является обработчик из 40-й строки. Но там стоит ограничение, позволяющее обрабатывать только исключения, вызванные переполнением. В результате обработка этого исключения передаётся в 48-ю строку, хотя этот обработчик дальше по стеку.

Листинг 11: Вложенные исключения

```
1  /* Task 6.
2    Nested exception process;
3    */
4
5  // IMPORTANT: Don't forget to disable Enhanced Instructions!!!
6  // Properties -> Configuration Properties -> C/C++ -> Code Generation ->
7  // Enable Enhanced Instruction Set = No Enhanced Instructions (/arch:IA32)
8
9  #include <stdio.h>
10 #include <tchar.h>
11 #include <cstring>
12 #include <cfloat>
13 #include <excpt.h>
14 #include <windows.h>
15 #include <time.h>
16
17 // log
18 FILE* logfile;
19
20 void initlog(const _TCHAR* prog);
21 void closelog();
22 void writelog(_TCHAR* format, ...);
23
24 // Defines the entry point for the console application.
25 int _tmain(int argc, _TCHAR* argv[]) {
26     //Init log
27     initlog(argv[0]);
28 }
```

```

29  // Floating point exceptions are masked by default.
30  _clearfp();
31  _controlfp_s(NULL, 0, _EM_OVERFLOW | _EM_ZERODIVIDE);
32
33  __try {
34      __try {
35          writelog(_T("Ready for generate DIVIDE_BY_ZERO exception.));
36          RaiseException(EXCEPTION_FLT_DIVIDE_BY_ZERO,
37              EXCEPTION_NONCONTINUABLE, 0, NULL);
38          writelog(_T("DIVIDE_BY_ZERO exception is generated.));
39      }
40      __except ((GetExceptionCode() == EXCEPTION_FLT_OVERFLOW) ?
41          EXCEPTION_EXECUTE_HANDLER :
42              EXCEPTION_CONTINUE_SEARCH)
43      {
44          writelog(_T("Internal handler in action.));
45          _tprintf(_T("Internal handler in action.));
46      }
47  }
48  __except ((GetExceptionCode() == EXCEPTION_FLT_DIVIDE_BY_ZERO) ?
49      EXCEPTION_EXECUTE_HANDLER :
50          EXCEPTION_CONTINUE_SEARCH)
51  {
52      writelog(_T("External handler in action.));
53      _tprintf(_T("External handler in action.));
54  }
55
56  closelog();
57  exit(0);
58 }
59
60 void initlog(const _TCHAR* prog) {
61     _TCHAR logname[255];
62     wcsncpy_s(logname, prog);
63
64     // replace extension
65     _TCHAR* extension;
66     extension = wcsstr(logname, _T(".exe"));
67     wcsncpy_s(extension, 5, _T(".log"), 4);
68
69     // Try to open log file for append
70     if (_wfopen_s(&logfile, logname, _T("a+"))) {
71         _wpperror(_T("The following error occurred));
72         _tprintf(_T("Can't open log file %s\n"), logname);
73         exit(1);

```

```

74     }
75
76     writelog(_T("%s is starting."), prog);
77 }
78
79 void closelog() {
80     writelog(_T("Shutting down.\n"));
81     fclose(logfile);
82 }
83
84 void writelog(_TCHAR* format, ...) {
85     _TCHAR buf[255];
86     va_list ap;
87
88     struct tm newtime;
89     __time64_t long_time;
90
91     // Get time as 64-bit integer.
92     _time64(&long_time);
93     // Convert to local time.
94     _localtime64_s(&newtime, &long_time);
95
96     // Convert to normal representation.
97     swprintf_s(buf, _T("[%d/%d/%d %d:%d:%d] "), newtime.tm_mday,
98         newtime.tm_mon + 1, newtime.tm_year + 1900, newtime.tm_hour,
99         newtime.tm_min, newtime.tm_sec);
100
101     // Write date and time
102     fwprintf(logfile, _T("%s"), buf);
103     // Write all params
104     va_start(ap, format);
105     _vsnwprintf_s(buf, sizeof(buf) - 1, format, ap);
106     fwprintf(logfile, _T("%s"), buf);
107     va_end(ap);
108     // New sting
109     fwprintf(logfile, _T("\n"));
110 }

```

Запуск отладчика подтвердил ожидаемый результат - поиск подходящего обработчика для исключения происходит снизу вверх. В начале проверяется ближайший обработчик (на рисунке 9 отрабатывает фильтр ближайшего обработчика исключения). Эта проверка вернёт EXCEPTION_CONTINUE_SEARCH для продолжения поиска более подходящего обработчика и передачи управления дальше по стеку.

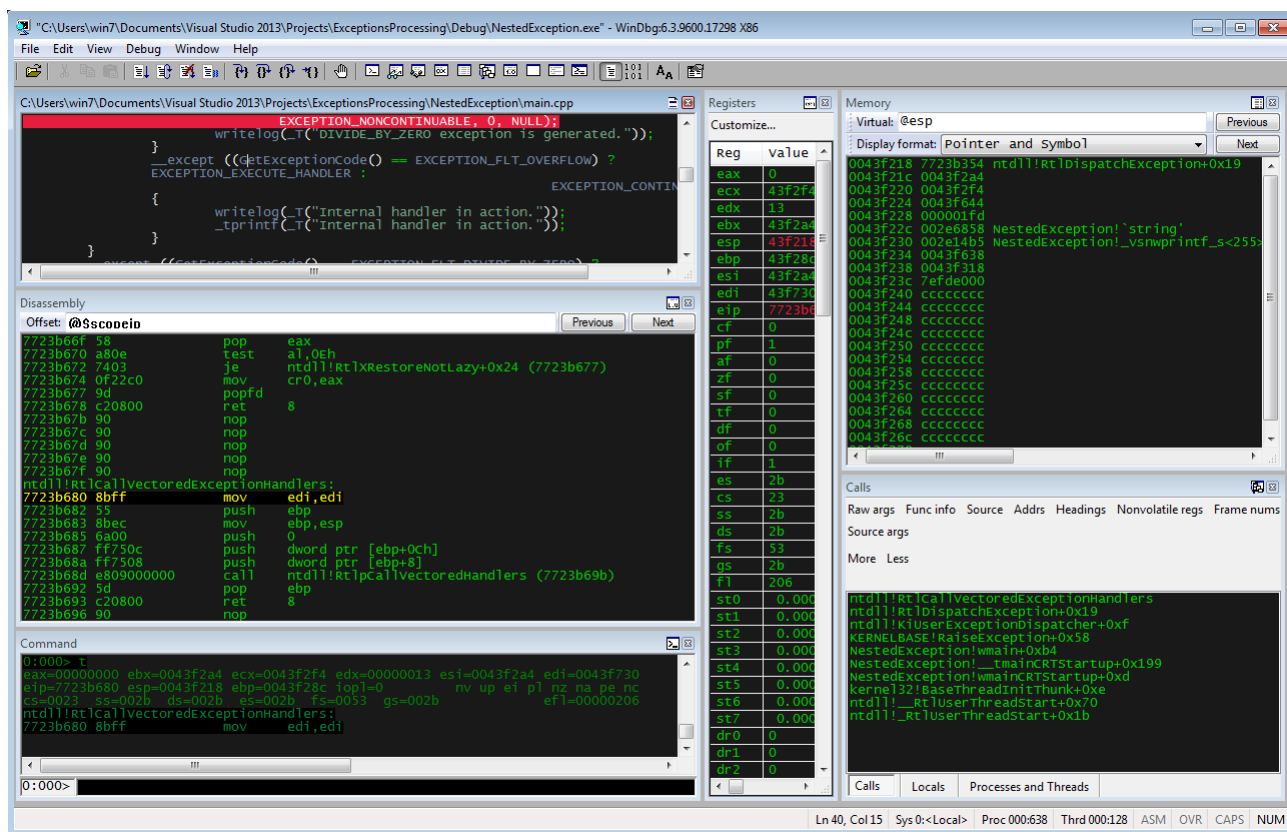


Рис. 9: Проверка условий обработчика исключения

При этом создаётся опасность утечки ресурсов, поэтому желательно обрабатывать исключительные ситуации в месте их возникновения. Протокол работы программы показан в листинге 12.

Листинг 12: Обработчик успел сохранить данные об исключении

- 1 [6/2/2015 18:43:54] C:\Users\win7\Documents\Visual Studio 2013\Projects\ ExceptionsProcessing\Debug\NestedException.exe is starting.
- 2 [6/2/2015 18:43:54] Ready for generate DIVIDE_BY_ZERO exception.
- 3 [6/2/2015 18:43:54] External handler in action.
- 4 [6/2/2015 18:43:54] Shutting down.

Выход при помощи goto

Использование goto считается дурной практикой по целому ряду причин. В листинге 13, благодаря goto управление со строки 35 передаётся сразу на строку 46. Таким образом осуществляется выход из блока __try без возбуждения и обработки исключения.

Листинг 13: Выход из блока охраняемого кода при помощи goto

```
1  /* Task 7.
2     Get out of the __try block by using the goto;
3     */
4
5  // IMPORTANT: Don't forget to disable Enhanced Instructions!!!
6  // Properties -> Configuration Properties -> C/C++ -> Code Generation ->
7  // Enable Enhanced Instruction Set = No Enhanced Instructions (/arch:IA32)
8
9  #include <stdio.h>
10 #include <tchar.h>
11 #include <cstring>
12 #include <cfloat>
13 #include <excpt.h>
14 #include <windows.h>
15 #include <time.h>
16
17 // log
18 FILE* logfile;
19
20 void initlog(const _TCHAR* prog);
21 void closelog();
22 void writelog(_TCHAR* format, ...);
23
24 // Defines the entry point for the console application.
25 int _tmain(int argc, _TCHAR* argv[]) {
26     //Init log
27     initlog(argv[0]);
28
29     // Floating point exceptions are masked by default.
30     _clearfp();
```



```

31  _controlfp_s(NULL, 0, _EM_OVERFLOW | _EM_ZERODIVIDE);
32
33  __try {
34      writelog(_T("Call goto"));
35      goto OUT_POINT;
36      writelog(_T("Ready for generate DIVIDE_BY_ZERO exception.));
37      RaiseException(EXCEPTION_FLT_DIVIDE_BY_ZERO,
38          EXCEPTION_NONCONTINUABLE, 0, NULL);
39      writelog(_T("DIVIDE_BY_ZERO exception is generated.));
40  }
41  __except (EXCEPTION_EXECUTE_HANDLER)
42  {
43      writelog(_T("Handler in action.));
44      _tprintf(_T("Handler in action.));
45  }
46 OUT_POINT:
47      writelog(_T("A point outside the __try block.));
48      _tprintf(_T("A point outside the __try block.));
49
50      closelog();
51      exit(0);
52 }
53
54 void initlog(const _TCHAR* prog) {
55     _TCHAR logname[255];
56     wcscpy_s(logname, prog);
57
58     // replace extension
59     _TCHAR* extension;
60     extension = wcsstr(logname, _T(".exe"));
61     wcsncpy_s(extension, 5, _T(".log"), 4);
62
63     // Try to open log file for append
64     if (_wfopen_s(&logfile, logname, _T("a+"))) {
65         _werror(_T("The following error occurred"));
66         _tprintf(_T("Can't open log file %s\n"), logname);
67         exit(1);
68     }
69
70     writelog(_T("%s is starting."), prog);
71 }
72
73 void closelog() {
74     writelog(_T("Shutting down.\n"));
75     fclose(logfile);

```

```

76 }
77
78 void writelog(_TCHAR* format, ...) {
79     _TCHAR buf[255];
80     va_list ap;
81
82     struct tm newtime;
83     __time64_t long_time;
84
85     // Get time as 64-bit integer.
86     _time64(&long_time);
87     // Convert to local time.
88     _localtime64_s(&newtime, &long_time);
89
90     // Convert to normal representation.
91     swprintf_s(buf, _T("[%d/%d/%d %d:%d:%d] "), newtime.tm_mday,
92         newtime.tm_mon + 1, newtime.tm_year + 1900, newtime.tm_hour,
93         newtime.tm_min, newtime.tm_sec);
94
95     // Write date and time
96     fwprintf(logfile, _T("%s"), buf);
97     // Write all params
98     va_start(ap, format);
99     _vsnwprintf_s(buf, sizeof(buf) - 1, format, ap);
100    fwprintf(logfile, _T("%s"), buf);
101    va_end(ap);
102    // New sting
103    fwprintf(logfile, _T("\n"));
104 }

```

На рисунке 10 видно, что как только достигнута строка с оператором goto, осуществляется безусловный переход к метке. Протокол работы программы (листинг 14) подтверждает, что до возбуждения исключения управление не дошло: после записи логге из 34-й строки идёт запись из 47-й, таким образом строки 36-39 пропущены.

Листинг 14: Переход по оператору goto

```

1 [6/2/2015 18:52:53] C:\Users\win7\Documents\Visual Studio 2013\Projects\
    ExceptionsProcessing\Debug\Goto.exe is starting.
2 [6/2/2015 18:52:53] Call goto
3 [6/2/2015 18:52:53] A point outside the __try block.
4 [6/2/2015 18:52:53] Shutting down.

```

Использование goto может привести к утечкам памяти в процессе раскрутки стека, в то же время он позволяет сделать переход сразу через несколько участков кода. Таким образом,

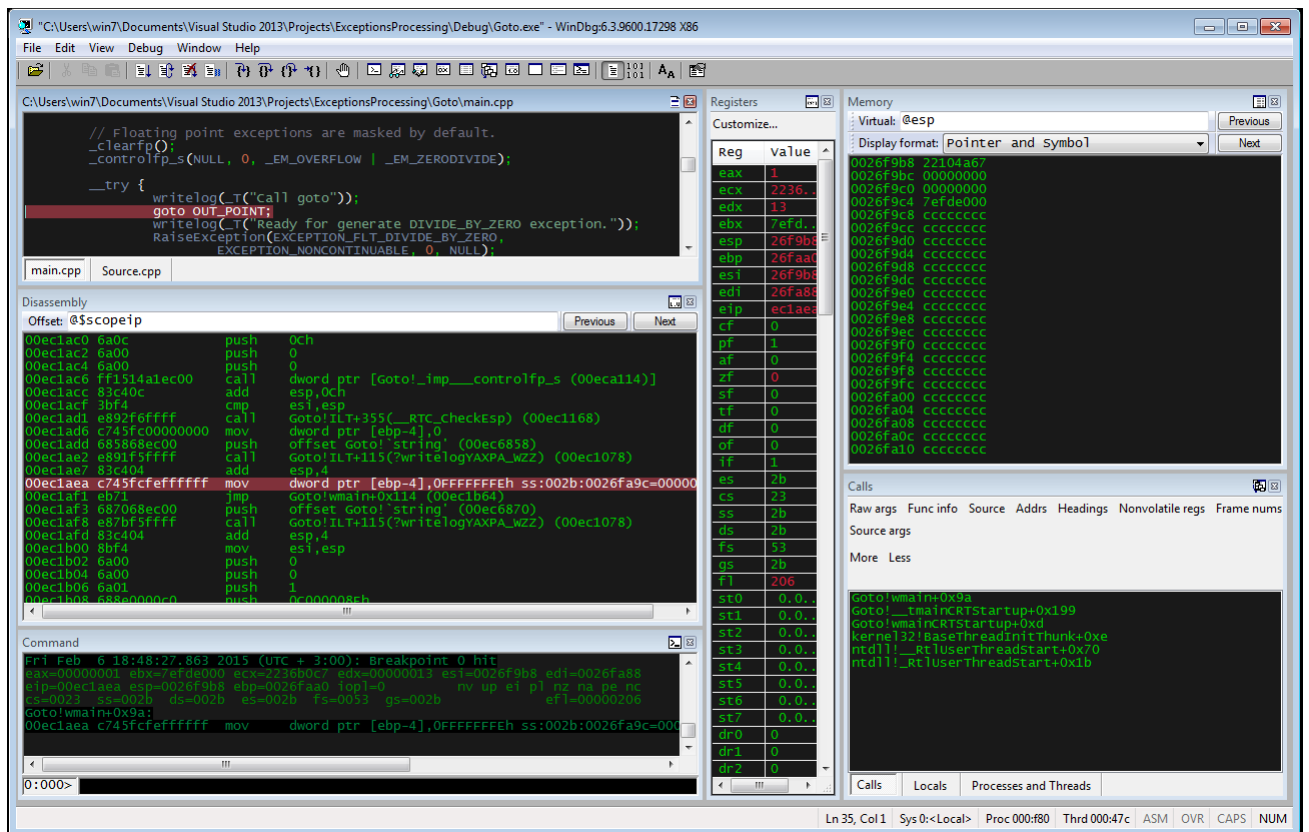


Рис. 10: Переход по goto

сфера применения `goto` достаточно узкая, и требует достаточно чёткого понимания.

Выход при помощи `__leave`

Листинг 15 похож на листинг 13, но за пределы охраняемого фрейма кода помогает выйти на этот раз `__leave`. Оператор `__leave` более эффективен, поскольку не вызывает разрушение стека.

Листинг 15: Выход из блока охраняемого кода при помощи `__leave`

```
1  /* Task 8.
2     Get out of the __try block by using the leave;
3     */
4
5  // IMPORTANT: Don't forget to disable Enhanced Instructions!!!
6  // Properties -> Configuration Properties -> C/C++ -> Code Generation ->
7  // Enable Enhanced Instruction Set = No Enhanced Instructions (/arch:IA32)
8
9  #include <stdio.h>
10 #include <tchar.h>
11 #include <cstring>
12 #include <cfloat>
13 #include <except.h>
14 #include <windows.h>
15 #include <time.h>
16
17 // log
18 FILE* logfile;
19
20 void initlog(const _TCHAR* prog);
21 void closelog();
22 void writelog(_TCHAR* format, ...);
23
24 // Defines the entry point for the console application.
25 int _tmain(int argc, _TCHAR* argv[]) {
26     //Init log
27     initlog(argv[0]);
28
29     // Floating point exceptions are masked by default.
30     _clearfp();
```

```

31 _controlfp_s(NULL, 0, _EM_OVERFLOW | _EM_ZERODIVIDE);
32
33 __try {
34     writelog(_T("Call __leave"));
35     __leave;
36     writelog(_T("Ready for generate DIVIDE_BY_ZERO exception.));
37     RaiseException(EXCEPTION_FLT_DIVIDE_BY_ZERO,
38         EXCEPTION_NONCONTINUABLE, 0, NULL);
39     writelog(_T("DIVIDE_BY_ZERO exception is generated.));
40 }
41 __except (EXCEPTION_EXECUTE_HANDLER)
42 {
43     writelog(_T("Handler in action.));
44     _tprintf(_T("Handler in action.));
45 }
46 writelog(_T("A point outside the __try block.));
47 _tprintf(_T("A point outside the __try block.));
48
49 closelog();
50 exit(0);
51 }
52
53 void initlog(const _TCHAR* prog) {
54     _TCHAR logname[255];
55     wcscpy_s(logname, prog);
56
57     // replace extension
58     _TCHAR* extension;
59     extension = wcsstr(logname, _T(".exe"));
60     wcsncpy_s(extension, 5, _T(".log"), 4);
61
62     // Try to open log file for append
63     if (_wfopen_s(&logfile, logname, _T("a+"))) {
64         _wprintf(_T("The following error occurred"));
65         _tprintf(_T("Can't open log file %s\n"), logname);
66         exit(1);
67     }
68
69     writelog(_T("%s is starting."), prog);
70 }
71
72 void closelog() {
73     writelog(_T("Shutting down.\n"));
74     fclose(logfile);
75 }

```

```

76
77 void writelog(_TCHAR* format, ...) {
78     _TCHAR buf[255];
79     va_list ap;
80
81     struct tm newtime;
82     __time64_t long_time;
83
84     // Get time as 64-bit integer.
85     _time64(&long_time);
86     // Convert to local time.
87     _localtime64_s(&newtime, &long_time);
88
89     // Convert to normal representation.
90     swprintf_s(buf, _T("[%d/%d/%d %d:%d:%d] "), newtime.tm_mday,
91         newtime.tm_mon + 1, newtime.tm_year + 1900, newtime.tm_hour,
92         newtime.tm_min, newtime.tm_sec);
93
94     // Write date and time
95     fwprintf(logfile, _T("%s"), buf);
96     // Write all params
97     va_start(ap, format);
98     _vsnwprintf_s(buf, sizeof(buf) - 1, format, ap);
99     fwprintf(logfile, _T("%s"), buf);
100    va_end(ap);
101    // New sting
102    fwprintf(logfile, _T("\n"));
103 }

```

Листинг 16: Переход по оператору `__leave`

```

1 [6/2/2015 19:8:37] C:\Users\win7\Documents\Visual Studio 2013\Projects\
   ExceptionsProcessing\Debug\Leave.exe is starting.
2 [6/2/2015 19:8:37] Call __leave
3 [6/2/2015 19:8:37] A point outside the __try block.
4 [6/2/2015 19:8:38] Shutting down.

```

Результат использования `__leave` — переход в конец блока `try` (грубо говоря, это можно рассматривать это как `goto` переход на закрывающую фигурную скобку блока `try` и вход в блок `finally` естественным образом). По сути, результат прежний (если смотреть на листинг 16), но метод его достижения отличается — этот способ считается более правильным, т.к. не приводит к раскрутке стека.

После перехода выполняется обработчик завершения. Хотя для получения того же результата можно использовать оператор `goto`, он (оператор `goto`) приводит к освобождению

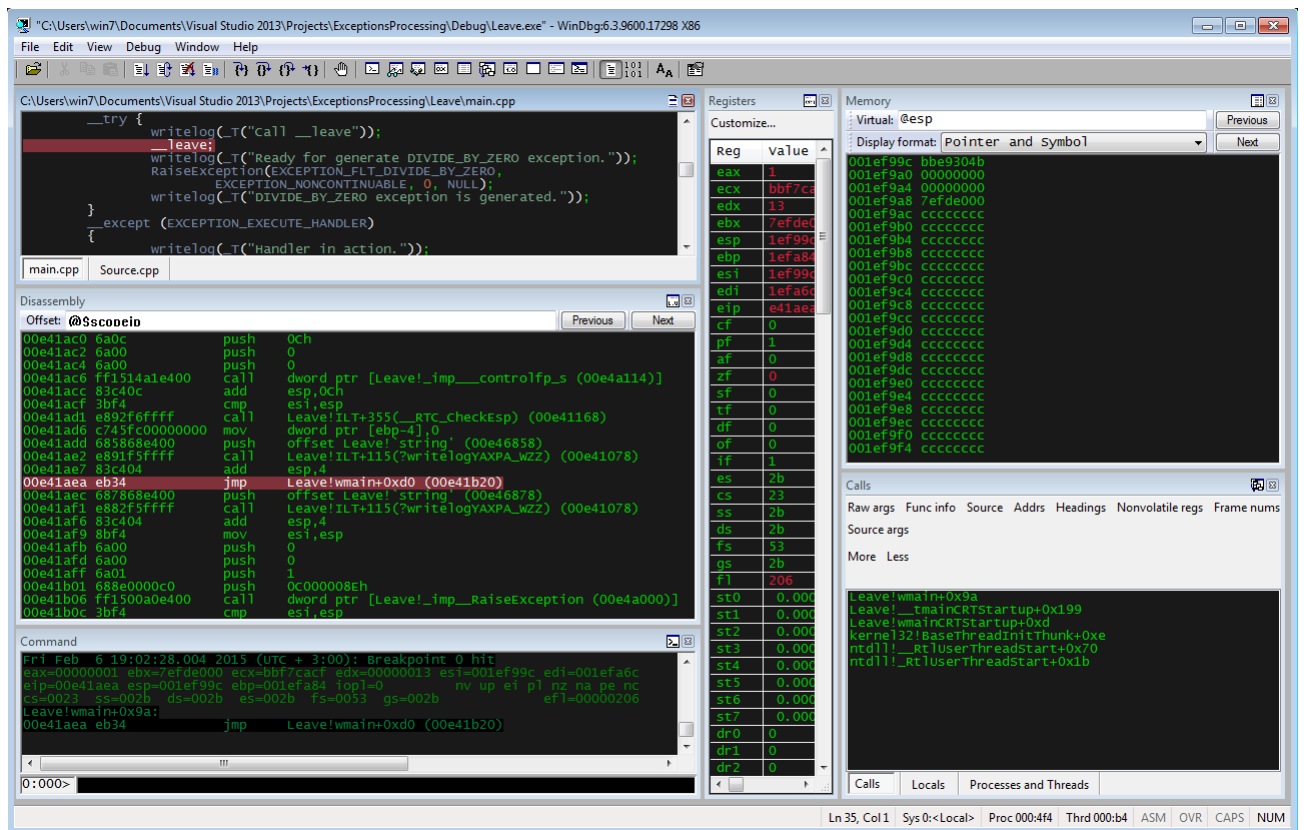


Рис. 11: Переход по __leave

стека. Одним из применений этого оператора является трассировка программ.

Преобразование SEH в C++

ИСКЛЮЧЕНИЕ

Листинг 17 показывает встраивание SEH в механизм исключений C/C++. Для этого необходимо включить соответствующие опции в компиляторе (/EHa).

Листинг 17: Трансформация исключений

```
1  /* Task 9.
2     Convert structural exceptions to the C language exceptions,
3     using the translator;
4     */
5
6  // IMPORTANT: Don't forget to disable Enhanced Instructions!!!
7  // Properties -> Configuration Properties -> C/C++ -> Code Generation ->
8  // Enable Enhanced Instruction Set = No Enhanced Instructions (/arch:IA32)
9
10 // IMPORTANT: Don't forget to enable SEH!!!
11 // Properties -> Configuration Properties -> C/C++ -> Code Generation ->
12 // Enable C++ Exceptions = Yes with SEH Exceptions (/EHa)
13
14 #include <stdio.h>
15 #include <tchar.h>
16 #include <cstring>
17 #include <cfloat>
18 #include <stdexcept>
19 #include <except.h>
20 #include <windows.h>
21 #include <time.h>
22
23 // log
24 FILE* logfile;
25
26 void initlog(const _TCHAR* prog);
27 void closelog();
28 void writelog(_TCHAR* format, ...);
29
```



```

30 void translator(unsigned int u, EXCEPTION_POINTERS* pExp);
31
32 // My exception
33 class translator_exception {
34 public:
35     translator_exception(const wchar_t* str) {
36         wcsncpy_s(buf, sizeof(buf), str, sizeof(buf));
37     }
38     const wchar_t* what() { return buf; }
39 private:
40     wchar_t buf[255];
41 };
42
43 // Defines the entry point for the console application.
44 int _tmain(int argc, _TCHAR* argv[]) {
45     //Init log
46     initlog(argv[0]);
47
48     // Floating point exceptions are masked by default.
49     _clearfp();
50     _controlfp_s(NULL, 0, _EM_OVERFLOW | _EM_ZERODIVIDE);
51
52     try {
53         writelog(_T("Ready for translator ativation."));
54         _set_se_translator(translator);
55         writelog(_T("Ready for generate DIVIDE_BY_ZERO exception."));
56         RaiseException(EXCEPTION_FLT_DIVIDE_BY_ZERO,
57             EXCEPTION_NONCONTINUABLE, 0, NULL);
58         writelog(_T("DIVIDE_BY_ZERO exception is generated."));
59     }
60     catch (translator_exception &e) {
61         _tprintf(_T("CPP exception: %s"), e.what());
62         writelog(_T("CPP exception: %s"), e.what());
63     }
64
65     closelog();
66     exit(0);
67 }
68
69 void translator(unsigned int u, EXCEPTION_POINTERS* pExp) {
70     writelog(_T("Translator in action."));
71     if (u == EXCEPTION_FLT_DIVIDE_BY_ZERO)
72         throw translator_exception(_T("EXCEPTION_FLT_DIVIDE_BY_ZERO"));
73 }
74

```

```

75 void initlog(const _TCHAR* prog) {
76     _TCHAR logname[255];
77     wcsncpy_s(logname, prog);
78
79     // replace extension
80     _TCHAR* extension;
81     extension = wcsstr(logname, _T(".exe"));
82     wcsncpy_s(extension, 5, _T(".log"), 4);
83
84     // Try to open log file for append
85     if (_wfopen_s(&logfile, logname, _T("a+"))) {
86         _wprintf(_T("The following error occurred"));
87         _tprintf(_T("Can't open log file %s\n"), logname);
88         exit(1);
89     }
90
91     writelog(_T("%s is starting."), prog);
92 }
93
94 void closelog() {
95     writelog(_T("Shutting down.\n"));
96     fclose(logfile);
97 }
98
99 void writelog(_TCHAR* format, ...) {
100     _TCHAR buf[255];
101     va_list ap;
102
103     struct tm newtime;
104     __time64_t long_time;
105
106     // Get time as 64-bit integer.
107     _time64(&long_time);
108     // Convert to local time.
109     _localtime64_s(&newtime, &long_time);
110
111     // Convert to normal representation.
112     swprintf_s(buf, _T("[%d/%d/%d %d:%d:%d] "), newtime.tm_mday,
113         newtime.tm_mon + 1, newtime.tm_year + 1900, newtime.tm_hour,
114         newtime.tm_min, newtime.tm_sec);
115
116     // Write date and time
117     fwprintf(logfile, _T("%s"), buf);
118     // Write all params
119     va_start(ap, format);

```

```

120     _vsnwprintf_s(buf, sizeof(buf) - 1, format, ap);
121     fwprintf(logfile, _T("%s"), buf);
122     va_end(ap);
123     // New sting
124     fwprintf(logfile, _T("\n"));
125 }

```

На рисунке 12 видна передача управления от генерации исключения в ядре к созданию пользовательского исключения. Листинг 18 показывает, какие участки кода были задействованы и в каком порядке.

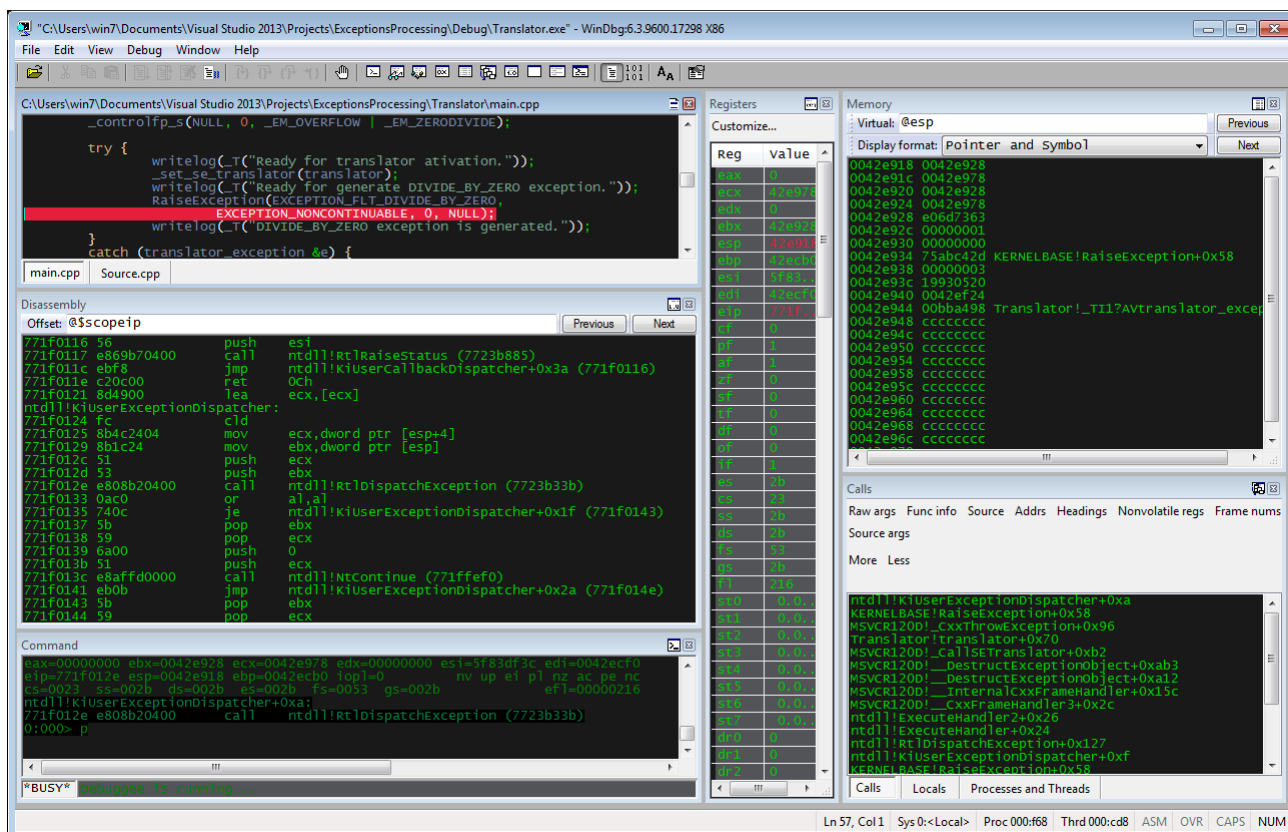


Рис. 12: Передача управления коду создания пользовательского исключения

Листинг 18: Результат работы Translator.exe

```

1 [6/2/2015 19:33:15] C:\Users\win7\Documents\Visual Studio 2013\Projects\
    ExceptionsProcessing\Debug\Translator.exe is starting.
2 [6/2/2015 19:33:15] Ready for translator ativation.
3 [6/2/2015 19:33:15] Ready for generate DIVIDE_BY_ZERO exception.
4 [6/2/2015 19:33:15] Translator in action.
5 [6/2/2015 19:33:15] CPP exception: EXCEPTION_FLT_DIVIDE_BY_ZERO
6 [6/2/2015 19:33:15] Shutting down.

```

Если проследить за передачей управления по стеку вызовов, то сразу после возбуждения исключения в 56-й строке, управление передаётся транслятору, где генерируется привычное

C++-исключения (в данном случае используется собственный класс исключения, определённый в 33-й строке), и только после этого в блок `catch`, где происходит обработка исключения.

Этот механизм способен обеспечить взаимодействие SEH с другими языками и системами.

Финальный обработчик finally

В листинге 19 исключение как таковое отсутствует, но есть охраняемый блок кода, и блок `__finally`, управление в который будет передано в любой ситуации.

Листинг 19: Исполнение кода в блоке `__finally`

```
1  /* Task 10.
2     Use the final handler finally;
3     */
4
5  // IMPORTANT: Don't forget to disable Enhanced Instructions!!!
6  // Properties -> Configuration Properties -> C/C++ -> Code Generation ->
7  // Enable Enhanced Instruction Set = No Enhanced Instructions (/arch:IA32)
8
9  #include <stdio.h>
10 #include <tchar.h>
11 #include <cfloat>
12 #include <excpt.h>
13 #include <time.h>
14 #include <windows.h>
15
16 // log
17 FILE* logfile;
18
19 void initlog(const _TCHAR* prog);
20 void closelog();
21 void writelog(_TCHAR* format, ...);
22
23 // Defines the entry point for the console application.
24 int _tmain(int argc, _TCHAR* argv[]) {
25     //Init log
26     initlog(argv[0]);
27
28     // Floating point exceptions are masked by default.
29     _clearfp();
30     _controlfp_s(NULL, 0, _EM_OVERFLOW | _EM_ZERODIVIDE);
31
```

```

32  __try {
33      // No exception
34      writelog(_T("Try block.));
35  }
36  __finally
37  {
38      writelog(_T("There is no exception, but the handler is called.));
39      _tprintf(_T("There is no exception, but the handler is called.));
40  }
41
42  closelog();
43  exit(0);
44 }
45
46 void initlog(const _TCHAR* prog) {
47     _TCHAR logname[255];
48     wcsncpy_s(logname, prog);
49
50     // replace extension
51     _TCHAR* extension;
52     extension = wcsstr(logname, _T(".exe"));
53     wcsncpy_s(extension, 5, _T(".log"), 4);
54
55     // Try to open log file for append
56     if (_wfopen_s(&logfile, logname, _T("a+"))) {
57         _wprintf(_T("The following error occurred"));
58         _tprintf(_T("Can't open log file %s\n"), logname);
59         exit(1);
60     }
61
62     writelog(_T("%s is starting."), prog);
63 }
64
65 void closelog() {
66     writelog(_T("Shutting down.\n"));
67     fclose(logfile);
68 }
69
70 void writelog(_TCHAR* format, ...) {
71     _TCHAR buf[255];
72     va_list ap;
73
74     struct tm newtime;
75     __time64_t long_time;
76

```


Листинг 20: Результат работы Finally.exe

```
1 [6/2/2015 19:42:27] C:\Users\win7\Documents\Visual Studio 2013\Projects\  
   ExceptionsProcessing\Debug\Finally.exe is starting.  
2 [6/2/2015 19:42:27] Try block.  
3 [6/2/2015 19:42:27] There is no exception, but the handler is called.  
4 [6/2/2015 19:42:27] Shutting down.
```

Вместо передачи управления обратно в программу, управление передаётся в блок `__finally` (рисунок 13). Более того, управление туда будет передано даже если блок защищаемого кода будет пуст. Листинг 20 показывает порядок исполнения кода.

Похожие механизмы есть в других распространённых языках программирования, они позволяют обеспечить строгие гарантии исключений, и не допустить нахождение объекта в неконсистентном состоянии.

Использование функции AbnormalTermination

В листинге 21 сравниваются два механизма из блока `__try`. Благодаря тому, что управление будет передано блоку `__finally` в любом случае, оказывается удобно в этом блоке проверять корректность выхода из блока `__try` (при помощи функции `AbnormalTermination`), и, в случае необходимости, корректно освобождать захваченные ресурсы.

Листинг 21: Проверка корректности выхода из блока `__try`

```
1  /* Task 11.  
2  Check the correctness of the exit from the __try block using  
3  the AbnormalTermination function in the final handler finally.  
4  */  
5  
6  // IMPORTANT: Don't forget to disable Enhanced Instructions!!!  
7  // Properties -> Configuration Properties -> C/C++ -> Code Generation ->  
8  // Enable Enhanced Instruction Set = No Enhanced Instructions (/arch:IA32)  
9  
10 #include <stdio.h>  
11 #include <tchar.h>  
12 #include <cstring>  
13 #include <cfloat>  
14 #include <excpt.h>  
15 #include <windows.h>  
16 #include <time.h>  
17  
18 // log  
19 FILE* logfile;  
20  
21 void initlog(const _TCHAR* prog);  
22 void closelog();  
23 void writelog(_TCHAR* format, ...);  
24  
25 // Defines the entry point for the console application.  
26 int _tmain(int argc, _TCHAR* argv[]) {
```

```

27 //Init log
28 initlog(argv[0]);
29
30 // Floating point exceptions are masked by default.
31 _clearfp();
32 _controlfp_s(NULL, 0, _EM_OVERFLOW | _EM_ZERODIVIDE);
33
34 __try {
35     writelog(_T("Call goto"));
36     goto OUT_POINT;
37     writelog(_T("Ready for generate DIVIDE_BY_ZERO exception.));
38     RaiseException(EXCEPTION_FLT_DIVIDE_BY_ZERO,
39         EXCEPTION_NONCONTINUABLE, 0, NULL);
40     writelog(_T("DIVIDE_BY_ZERO exception is generated.));
41 }
42 __finally
43 {
44     if (AbnormalTermination()) {
45         writelog(_T("%s"), _T("Abnormal termination in goto case.));
46         _tprintf(_T("%s"), _T("Abnormal termination in goto case.\n"));
47     }
48     else {
49         writelog(_T("%s"), _T("Normal termination in goto case.));
50         _tprintf(_T("%s"), _T("Normal termination in goto case.\n"));
51     }
52 }
53 OUT_POINT:
54 writelog(_T("A point outside the first __try block.));
55
56 __try {
57     writelog(_T("Call __leave"));
58     __leave;
59     writelog(_T("Ready for generate EXCEPTION_FLT_DIVIDE_BY_ZERO exception."
60         ));
61     RaiseException(EXCEPTION_FLT_DIVIDE_BY_ZERO,
62         EXCEPTION_NONCONTINUABLE, 0, NULL);
63     writelog(_T("EXCEPTION_FLT_DIVIDE_BY_ZERO exception is generated.));
64 }
65 __finally
66 {
67     if (AbnormalTermination()) {
68         writelog(_T("%s"), _T("Abnormal termination in __leave case.));
69         _tprintf(_T("%s"), _T("Abnormal termination in __leave case.\n"));
70     }
71     else {

```

```

71     writelog(_T("%s"), _T("Normal termination in __leave case.));
72     _tprintf(_T("%s"), _T("Normal termination in __leave case.\n"));
73 }
74 }
75 writelog(_T("A point outside the second __try block.));
76
77 closelog();
78 exit(0);
79 }
80
81 void initlog(const _TCHAR* prog) {
82     _TCHAR logname[255];
83     wcsncpy_s(logname, prog);
84
85     // replace extension
86     _TCHAR* extension;
87     extension = wcsstr(logname, _T(".exe"));
88     wcsncpy_s(extension, 5, _T(".log"), 4);
89
90     // Try to open log file for append
91     if (_wfopen_s(&logfile, logname, _T("a+"))) {
92         _wprintf(_T("The following error occurred"));
93         _tprintf(_T("Can't open log file %s\n"), logname);
94         exit(1);
95     }
96
97     writelog(_T("%s is starting."), prog);
98 }
99
100 void closelog() {
101     writelog(_T("Shutting down.\n"));
102     fclose(logfile);
103 }
104
105 void writelog(_TCHAR* format, ...) {
106     _TCHAR buf[255];
107     va_list ap;
108
109     struct tm newtime;
110     __time64_t long_time;
111
112     // Get time as 64-bit integer.
113     _time64(&long_time);
114     // Convert to local time.
115     _localtime64_s(&newtime, &long_time);

```

```

116
117 // Convert to normal representation.
118 swprintf_s(buf, _T("[%d/%d/%d %d:%d:%d] "), newtime.tm_mday,
119         newtime.tm_mon + 1, newtime.tm_year + 1900, newtime.tm_hour,
120         newtime.tm_min, newtime.tm_sec);
121
122 // Write date and time
123 fwprintf(logfile, _T("%s"), buf);
124 // Write all params
125 va_start(ap, format);
126 _vsnwprintf_s(buf, sizeof(buf) - 1, format, ap);
127 fwprintf(logfile, _T("%s"), buf);
128 va_end(ap);
129 // New sting
130 fwprintf(logfile, _T("\n"));
131 }

```

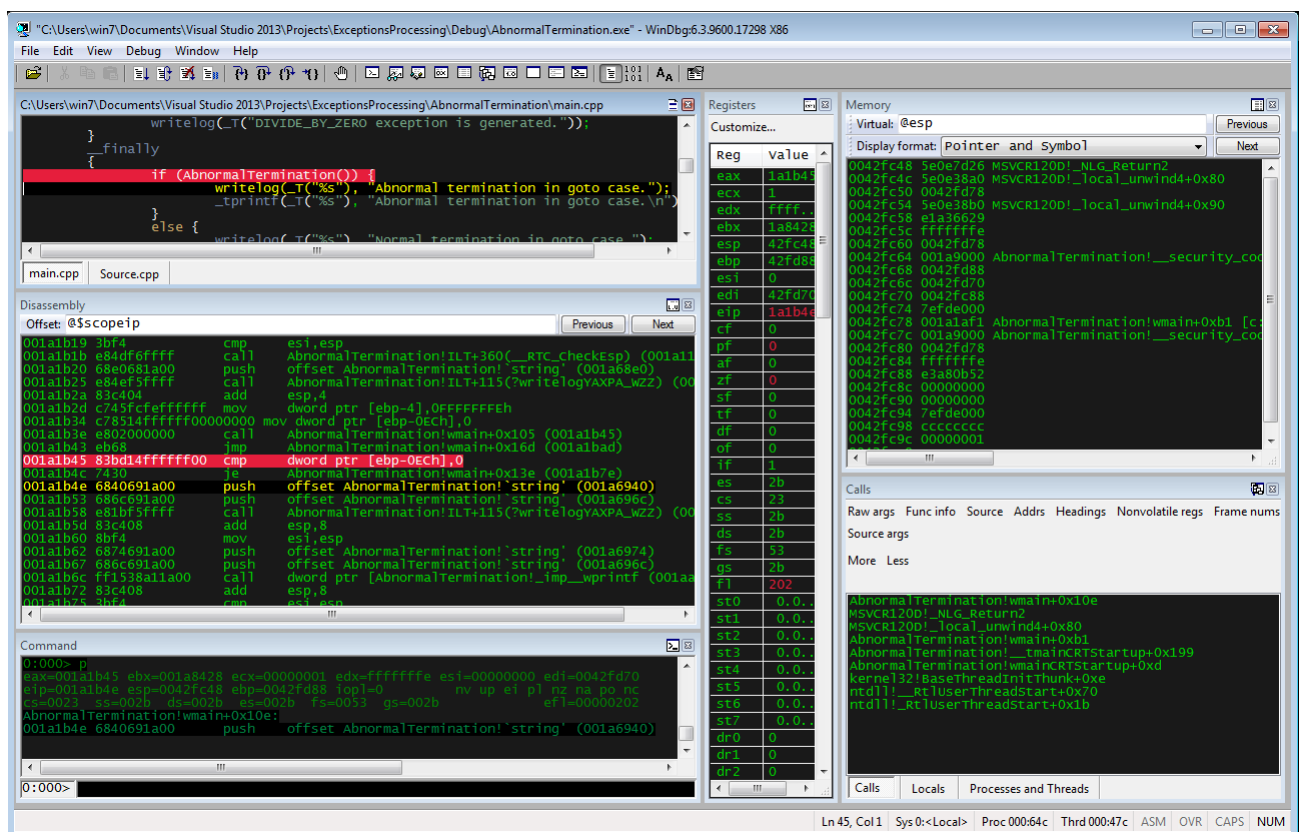


Рис. 14: Выход из защищаемого блока по goto

Функция `AbnormalTermination()` позволяет определить на сколько правильным был выход из защищаемого кода в случае с `goto` (рисунок 14) и `__leave` (рисунок 15). Протокол работы программы представлен в листинге 22.

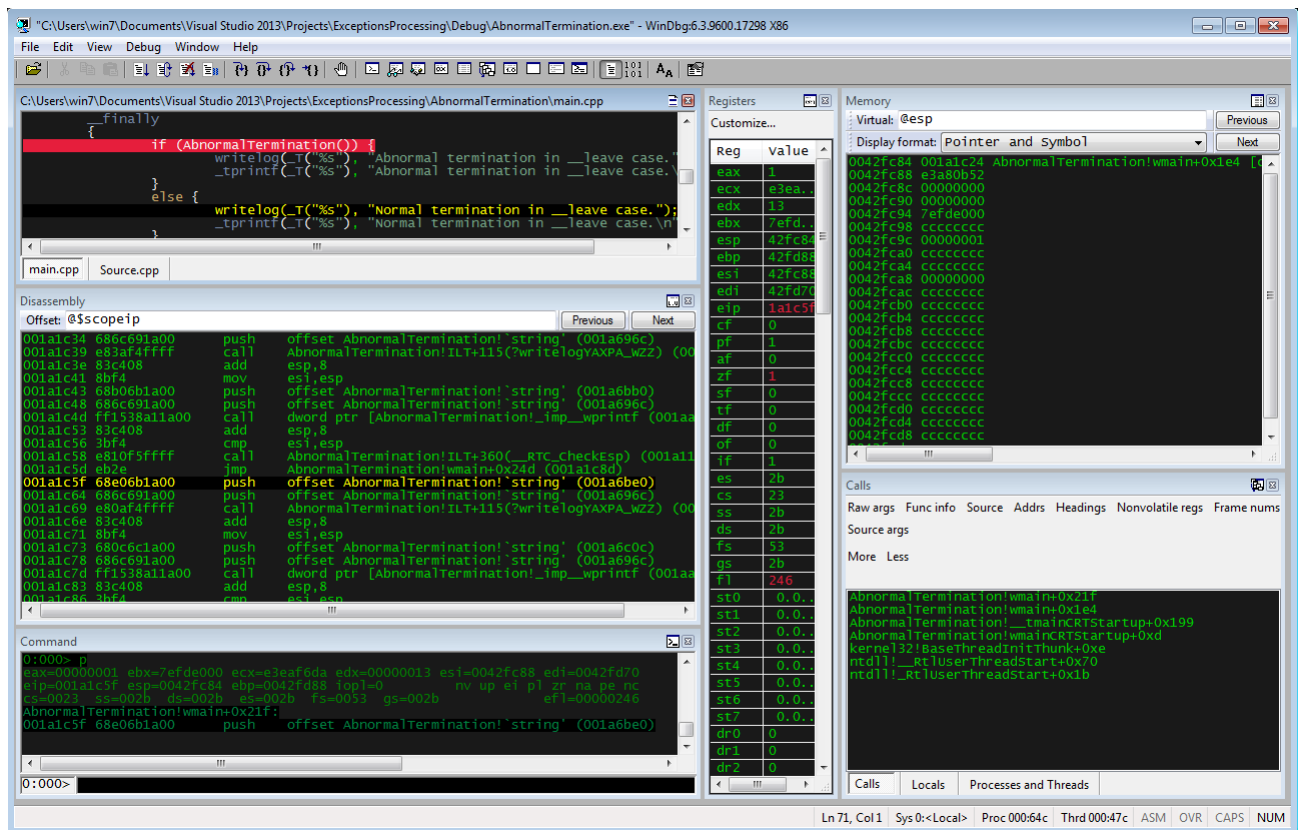


Рис. 15: Выход из защищаемого блока по `__leave`

В зависимости от этого принимается решение об освобождении захваченных ресурсов, но если по какой-то причине нужно выйти из защищаемого блока (хотя причина такой необходимости не очевидна) лучше использовать `__leave`, т.к. с `goto` больше шансов на утечку ресурсов, захваченных (и не освобождённых) в блоке `__try`.

Листинг 22: Результат работы Finally.exe

```

1 [6/2/2015 19:58:19] C:\Users\win7\Documents\Visual Studio 2013\Projects\
  ExceptionsProcessing\Debug\AbnormalTermination.exe is starting.
2 [6/2/2015 19:58:19] Call goto
3 [6/2/2015 19:58:19] Abnormal termination in goto case.
4 [6/2/2015 19:58:19] A point outside the first __try block.
5 [6/2/2015 19:58:19] Call __leave
6 [6/2/2015 19:58:19] Normal termination in __leave case.
7 [6/2/2015 19:58:19] A point outside the second __try block.
8 [6/2/2015 19:58:19] Shutting down.

```

Заключение

При обработке исключений в C++ используются ключевые слова `catch` и `throw`, а сам механизм исключений реализован с использованием SEH. Тем не менее, обработка исключений в C++ и SEH — это разные вещи. Их совместное применение требует внимательного обращения, поскольку обработчики исключений, написанные пользователем и сгенерированные C++, могут взаимодействовать между собой и приводить к нежелательным последствиям. Документация Microsoft рекомендует полностью отказаться от использования обработчиков Windows в прикладных программах на C++ и ограничиться применением в них только обработчиков исключений C++.

Кроме того, обработчики исключений или завершения Windows не осуществляют вызов деструкторов, что в ряде случаев необходимо для уничтожения экземпляров объектов C++.

В то же время, наличие таких мощных инструментов как блок `__finally`, гибкая система фильтрации и извлечение контекста исключения делает их незаменимыми при разработке системного ПО.

Таким образом, нужно чётко понимать, что механизм SEH и исключения, реализованные на уровне языка C++ это разные инструменты, требующие разного подхода.