Санкт-Петербургский государственный политехнический университет
Институт Информационных Технологий и Управления
Кафедра компьютерных систем и программных технологий

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.

Санкт-Петербург
2014

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.

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

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. 48-40); кроме того, операции с плавающими точками выполняются асинхронно, и нужно на этапе компиляции отключить расширения векторизации.

В 53-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.

Listing 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
17 void usage(const _TCHAR *prog);
18
19 // Task switcher
20 enum {
21     DIVIDE_BY_ZERO,
22     FLT_OVERFLOW
23 } task;
24
25 // Defines the entry point for the console application.
26 int _tmain(int argc, _TCHAR* argv[]) {
27     // Check parameters number
28     if (argc != 2) {
29         printf("Too few parameters.\n\n");
30         usage(argv[0]);
31         return 1;
32     }
33
34     // Set task
35     if (!_tcscmp(_T("-d"), argv[1])) {
36         task = DIVIDE_BY_ZERO;
37     }
38     else if (!_tcscmp(_T("-o"), argv[1])) {
39         task = FLT_OVERFLOW;
40     }
41     else {
42         printf("Can't parse parameters.\n\n");
43         usage(argv[0]);
44         return 2;
```

```
45      }
46
47      // Floating point exceptions are masked by default.
48      _clearfp();
49      _controlfp_s(NULL, 0, _EM_OVERFLOW | _EM_ZERODIVIDE);
50
51      // Set exception
52      __try {
53          volatile float tmp = 0;
54          switch (task) {
55          case DIVIDE_BY_ZERO:
56              tmp = 1 / tmp;
57              break;
58          case FLT_OVERFLOW:
59              // Note: floating point execution happens asynchronously.
60              // So, the exception will not be handled until the next floating
61              // point instruction.
62              tmp = pow(FLT_MAX, 3);
63              break;
64          default:
65              break;
66          }
67      }
68      __except (EXCEPTION_EXECUTE_HANDLER) {
69          printf("Well, it looks like we caught something.");
70      }
71      return 0;
72  }
73
74  // Usage manual
75  void usage(const _TCHAR *prog) {
76      printf("Usage: \n");
77      _tprintf(_T("\t%s -d\n"), prog);
78      printf("\t\t\t for exception float divide by zero,\n");
79      _tprintf(_T("\t%s -o\n"), prog);
80      printf("\t\t\t for exception float overflow.\n");
81  }
```

Если запустить этот код в отладчике, и пройти его по шагам, то можно увидеть, как управление передаётся с 56-й строки (либо 62-й, в зависимости от параметров, переданных при запуске) передаётся на 68-ю. Фильтр отсутствует, так что обработка сразу переходит на 69-ю строку и дальше до конца программы (до выхода, т.е. обратной передачи управления не происходит).

В листинге 2 показано использование функции GetExceptionCode. В первом случае она участвует в сравнении с макро-константной EXCEPTION_FLT_DIVIDE_BY_ZERO для определения подходящего обработчика для исключительного события. Во-втором случае она используется уже внутри обработчика, позволяя определить, что исключение вызвано переполнением при операции с типом float.

Listing 2: Получение кода исключения с помощью функции GetExceptionCode

```
/*  Task 2.
    Get the exceptions code using the GetExceptionCode gunction:
    - Use this function in the filter expression;
    - Use this function in the handler.
*/

// IMPORTANT: Don't forget to disable Enhanced Instructions!!!
// Properties -> Configuration Properties -> C/C++ -> Code Generation ->
// Enable Enhanced Instruction Set = No Enhanced Instructions (/arch:IA32)

#include <stdio.h>
#include <tchar.h>
#include <cstring>
#include <cfloat>
#include <cmath>
#include <excpt.h>
#include <windows.h>

void usage(const _TCHAR *prog);

// Task switcher
enum {
    DIVIDE_BY_ZERO,
    FLT_OVERFLOW
} task;

// Defines the entry point for the console application.
int _tmain(int argc, _TCHAR* argv[]) {
    // Check parameters number
    if (argc != 2) {
        printf("Too few parameters.\n\n");
        usage(argv[0]);
        return 1;
    }

    // Set task
    if (!_tcscmp(_T("-d"), argv[1])) {
        task = DIVIDE_BY_ZERO;
    }
    else if (!_tcscmp(_T("-o"), argv[1])) {
        task = FLT_OVERFLOW;
    }
    else {
        printf("Can't parse parameters.\n\n");
        usage(argv[0]);
        return 2;
    }

    // Floating point exceptions are masked by default.
```

```
50      _clearfp();
51      _controlfp_s(NULL, 0, _EM_OVERFLOW | _EM_ZERODIVIDE);
52
53      // Set exception
54      volatile float tmp = 0;
55      switch (task) {
56      case DIVIDE_BY_ZERO:
57          __try {
58              tmp = 1 / tmp;
59          }
60          // Use GetExceptionCode() function in the filter expression;
61          __except ((GetExceptionCode() == EXCEPTION_FLT_DIVIDE_BY_ZERO) ?
62          EXCEPTION_EXECUTE_HANDLER :
63                                          EXCEPTION_CONTINUE_SEARCH)
64          {
65              printf("Caught exception is: EXCEPTION_FLT_DIVIDE_BY_ZERO");
66          }
67          break;
68      case FLT_OVERFLOW:
69          __try {
70              // Note: floating point execution happens asynchronously.
71              // So, the exception will not be handled until the next
72              // floating point instruction.
73              tmp = pow(FLT_MAX, 3);
74          }
75          // Use GetExceptionCode() function in the handler.
76          __except (EXCEPTION_EXECUTE_HANDLER) {
77              if (GetExceptionCode() == EXCEPTION_FLT_OVERFLOW)
78                  printf("Caught exception is: EXCEPTION_FLT_OVERFLOW");
79              else
80                  printf("UNKNOWN exception: %x\n", GetExceptionCode());
81          }
82          break;
83      default:
84          break;
85      }
86      return 0;
87  }
88
89  // Usage manual
90  void usage(const _TCHAR *prog) {
91      printf("Usage: \n");
92      _tprintf(_T("\t%s -d\n"), prog);
93      printf("\t\t\t for exception float divide by zero,\n");
94      _tprintf(_T("\t%s -o\n"), prog);
95      printf("\t\t\t for exception float overflow.\n");
96  }
```

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

Запуск под отладчиком позволяет проследить передачу управления через стек вызовов.

В листинге 3 представлена функция-фильтр, которая возвращает EXCEPTION_CONTINUE_SEARCH только если исключение вызвано EXCEPTION_FLT_DIVIDE_BY_ZERO или EXCEPTION_FLT_OVERFLOW.

Listing 3: 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
17 void usage(const _TCHAR *prog);
18 LONG Filter(DWORD dwExceptionGode);
19
20 // Task switcher
21 enum {
22     DIVIDE_BY_ZERO,
23     FLT_OVERFLOW
24 } task;
25
26 // Defines the entry point for the console application.
27 int _tmain(int argc, _TCHAR* argv[]) {
28     // Check parameters number
29     if (argc != 2) {
30         printf("Too few parameters.\n\n");
31         usage(argv[0]);
32         return 1;
33     }
34
35     // Set task
36     if (!_tcscmp(_T("-d"), argv[1])) {
37         task = DIVIDE_BY_ZERO;
38     }
39     else if (!_tcscmp(_T("-o"), argv[1])) {
40         task = FLT_OVERFLOW;
41     }
42     else {
43         printf("Can't parse parameters.\n\n");
44         usage(argv[0]);
45         return 2;
46     }
47
48     // Floating point exceptions are masked by default.
49     _clearfp();
50     _controlfp_s(NULL, 0, _EM_OVERFLOW | _EM_ZERODIVIDE);
51
```

```
52      // Set exception
53      __try {
54          volatile float tmp = 0;
55          switch (task) {
56          case DIVIDE_BY_ZERO:
57              tmp = 1 / tmp;
58              break;
59          case FLT_OVERFLOW:
60              // Note: floating point execution happens asynchronously.
61              // So, the exception will not be handled until the next floating
62              // point instruction.
63              tmp = pow(FLT_MAX, 3);
64              break;
65          default:
66              break;
67          }
68      }
69      // Own filter function.
70      __except (Filter(GetExceptionCode())) {
71          printf("Caught exception is: ");
72          switch (GetExceptionCode()){
73          case EXCEPTION_FLT_DIVIDE_BY_ZERO:
74              printf("EXCEPTION_FLT_DIVIDE_BY_ZERO"); break;
75          case EXCEPTION_FLT_OVERFLOW:
76              printf("EXCEPTION_FLT_OVERFLOW"); break;
77          default:
78              printf("UNKNOWN exception: %x\n", GetExceptionCode()); break;
79          }
80      }
81      return 0;
82  }
83
84  // Own filter function.
85  LONG Filter(DWORD dwExceptionGode) {
86      if (dwExceptionGode == EXCEPTION_FLT_DIVIDE_BY_ZERO ||
87                  dwExceptionGode == EXCEPTION_FLT_OVERFLOW)
88          return EXCEPTION_EXECUTE_HANDLER;
89      return EXCEPTION_CONTINUE_SEARCH;
90  }
91
92  // Usage manual
93  void usage(const _TCHAR *prog) {
94      printf("Usage: \n");
95      _tprintf(_T("\t%s -d\n"), prog);
96      printf("\t\t\t for exception float divide by zero,\n");
97      _tprintf(_T("\t%s -o\n"), prog);
98      printf("\t\t\t for exception float overflow.\n");
99  }
```

При изучении работы программы под отладчиком, выяснилось что при возникновении исключение не передаётся в то место, где определена функция-фильтр. Вероятно компилятор оптимизирует код и подставляет её целиком на место вызова.

Листинг 4 показывает получение информации об исключении 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.

Кроме GetExceptionInformation, в листинге показан программный вызов исключений при помощи функции RaiseException. Она обладает 4-я параметрами, но наиболее важным является первый, который определяет тип возбуждаемого исключения.

Listing 4: Программная генерация исключения при помощи функции RaiseException

```
/*  Task 4.
    Get information about the exception using the GetExceptionInformation() fnc;
    throw an exception using the RaiseException() function.
*/

// IMPORTANT: Don't forget to disable Enhanced Instructions!!!
// Properties -> Configuration Properties -> C/C++ -> Code Generation ->
// Enable Enhanced Instruction Set = No Enhanced Instructions (/arch:IA32)

#include <stdio.h>
#include <tchar.h>
#include <cstring>
#include <cfloat>
#include <cmath>
#include <excpt.h>
#include <windows.h>

void usage(const _TCHAR *prog);
LONG Filter(DWORD dwExceptionGode, const _EXCEPTION_POINTERS *ep);

// Task switcher
enum {
    DIVIDE_BY_ZERO,
    FLT_OVERFLOW
} task;

// Defines the entry point for the console application.
int _tmain(int argc, _TCHAR* argv[]) {
    // Check parameters number
    if (argc != 2) {
        printf("Too few parameters.\n\n");
        usage(argv[0]);
        return 1;
    }

    // Set task
    if (!_tcscmp(_T("-d"), argv[1])) {
        task = DIVIDE_BY_ZERO;
    }
    else if (!_tcscmp(_T("-o"), argv[1])) {
        task = FLT_OVERFLOW;
    }
    else {
        printf("Can't parse parameters.\n\n");
        usage(argv[0]);
```

```
46          return 2;
47      }
48
49      // Floating point exceptions are masked by default.
50      _clearfp();
51      _controlfp_s(NULL, 0, _EM_OVERFLOW | _EM_ZERODIVIDE);
52
53      __try {
54          switch (task) {
55          case DIVIDE_BY_ZERO:
56              // throw an exception using the RaiseException() function
57              RaiseException(EXCEPTION_FLT_DIVIDE_BY_ZERO,
58                                                  EXCEPTION_NONCONTINUABLE, 0, NULL);
59              break;
60          case FLT_OVERFLOW:
61              // throw an exception using the RaiseException() function
62              RaiseException(EXCEPTION_FLT_OVERFLOW,
63                                                  EXCEPTION_NONCONTINUABLE, 0, NULL);
64              break;
65          default:
66              break;
67          }
68      }
69      __except (Filter(GetExceptionCode(), GetExceptionInformation())) {
70          // There is nothing to do, everything is done in the filter function.
71      }
72      return 0;
73  }
74
75  LONG Filter(DWORD dwExceptionGode,
76                              const _EXCEPTION_POINTERS *ExceptionPointers) {
77      enum { size = 200 };
78      char buf[size] = { '\0' };
79      const char* err = "Fatal error!\nexeption code: 0x";
80      const char* mes = "\nProgram terminate!";
81      if (ExceptionPointers)
82      // Get information about the exception using the GetExceptionInformation
83          sprintf_s(buf, "%s%x%s%x%s%x%s", err,
84                  ExceptionPointers->ExceptionRecord->ExceptionCode,
85                  ", data adress: 0x",
86                  ExceptionPointers->ExceptionRecord->ExceptionInformation[1],
87                  ", instruction adress: 0x",
88                  ExceptionPointers->ExceptionRecord->ExceptionAddress, mes);
89      else
90          sprintf_s(buf, "%s%x%s", err, dwExceptionGode, mes);
91      printf("%s", buf);
92
93      return EXCEPTION_EXECUTE_HANDLER;
94  }
95
96  // Usage manual
97  void usage(const _TCHAR *prog) {
98      printf("Usage: \n");
99      _tprintf(_T("\t%s -d\n"), prog);
100     printf("\t\t\t for exception float divide by zero,\n");
```

```
101         _tprintf(_T("\t%s -o\n"), prog);
102         printf("\t\t\t for exception float overflow.\n");
103     }
```

Важной особенностью функции GetExceptionlnformation 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.

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 всегда возвращает EXCEPTION_EXECUTE_HANDLER). Затем, в зависимости от настроек операционной системы, вызывается либо отладчик, либо функция NtRaiseHardError, которая отображает сообщение об ошибке.

Листинг 5 показывает работу с UnhandledExceptionFilter. Этот код лучше запускать под отладчиком, т.к. UnhandledExceptionFilter в конце вызывает EXCEPTION_CONTINUE_SEARCH, пытаясь передать управление именно отладчику (или вышестоящему по стеку обработчику, если бы он там был).

Listing 5: 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
18 void usage(const _TCHAR *prog);
19 LONG WINAPI UnhandledExceptionFilter(
20                                     const _EXCEPTION_POINTERS *ExceptionInfo);
21
22 // Task switcher
23 enum {
24     DIVIDE_BY_ZERO,
25     FLT_OVERFLOW
26 } task;
27
28 // Defines the entry point for the console application.
29 int _tmain(int argc, _TCHAR* argv[]) {
30     // Check parameters number
31     if (argc != 2) {
32         printf("Too few parameters.\n\n");
33         usage(argv[0]);
34         return 1;
35     }
36
37     // Set task
38     if (!_tcscmp(_T("-d"), argv[1])) {
39         task = DIVIDE_BY_ZERO;
40     }
41     else if (!_tcscmp(_T("-o"), argv[1])) {
```

```
42          task = FLT_OVERFLOW;
43      }
44      else {
45          printf("Can't parse parameters.\n\n");
46          usage(argv[0]);
47          return 2;
48      }
49
50      // Floating point exceptions are masked by default.
51      _clearfp();
52      _controlfp_s(NULL, 0, _EM_OVERFLOW | _EM_ZERODIVIDE);
53
54      volatile float tmp = 0;
55      SetUnhandledExceptionFilter(UnhandledExceptionFilter);
56
57      switch (task) {
58      case DIVIDE_BY_ZERO:
59          // throw an exception using the RaiseException() function
60          RaiseException(EXCEPTION_FLT_DIVIDE_BY_ZERO,
61                                                  EXCEPTION_EXECUTE_FAULT, 0, NULL);
62          break;
63      case FLT_OVERFLOW:
64          // throw an exception using the RaiseException() function
65          RaiseException(EXCEPTION_FLT_OVERFLOW,
66                                                  EXCEPTION_EXECUTE_FAULT, 0, NULL);
67          break;
68      default:
69          break;
70      }
71
72      return 0;
73  }
74
75  LONG WINAPI UnhandledExceptionFilter(
76                              const _EXCEPTION_POINTERS *ExceptionPointers) {
77      enum { size = 200 };
78      char buf[size] = { '\0' };
79      const char* err = "Unhandled exception!\nexeption code : 0x";
80      // Get information about the exception using the GetExceptionInformation
81      sprintf_s(buf, "%s%x%s%x%s%x%s", err,
82                      ExceptionPointers->ExceptionRecord->ExceptionCode,
83                      ", data adress: 0x",
84                      ExceptionPointers->ExceptionRecord->ExceptionInformation[1],
85                      ", instruction adress: 0x",
86                      ExceptionPointers->ExceptionRecord->ExceptionAddress);
87      printf("%s", buf);
88
89      return EXCEPTION_CONTINUE_SEARCH;
90  }
91
92  // Usage manual
93  void usage(const _TCHAR *prog) {
94      printf("Usage: \n");
95      _tprintf(_T("\t%s -d\n"), prog);
96      printf("\t\t\t for exception float divide by zero,\n");
```

```
97      _tprintf(_T("\t%s -o\n"), prog);
98      printf("\t\t\t for exception float overflow.\n");
99  }
```

При работе с отладчиком, я получал сообщение об ошибке в KernelBase.dll. Возможно, это связано с тем, что я отключил аппаратную поддержку, и какие-то операции выполнялись программно, на уровне ядра.

При запуске без отладчика, исключение долетало до самого верхнего уровня, операционная система сообщала об ошибке и сама предлагала запустить отладчик.

Листинг 6 показывает, как происходит передача исключения, в поисках подходящего обработчика. Самым ближайшим (по стеку) обработчиком для исключения, вызванного делением на 0, является обработчик из 29-й строки. Но там стоит ограничение, позволяющее обрабатывать только исключения, вызванные переполнением. В результате обработка этого исключения передаётся в 36-ю строку, хотя этот обработчик дальше по стеку.

Listing 6: 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
16 void usage(const _TCHAR *prog);
17
18 // Defines the entry point for the console application.
19 int _tmain(int argc, _TCHAR* argv[]) {
20     // Floating point exceptions are masked by default.
21     _clearfp();
22     _controlfp_s(NULL, 0, _EM_OVERFLOW | _EM_ZERODIVIDE);
23
24     __try {
25         __try {
26             RaiseException(EXCEPTION_FLT_DIVIDE_BY_ZERO,
27                                         EXCEPTION_NONCONTINUABLE, 0, NULL);
28         }
29         __except ((GetExceptionCode() == EXCEPTION_FLT_OVERFLOW) ?
30             EXCEPTION_EXECUTE_HANDLER :
31                                         EXCEPTION_CONTINUE_SEARCH)
32         {
33             printf("Internal handler in action.");
34         }
35     }
36     __except ((GetExceptionCode() == EXCEPTION_FLT_DIVIDE_BY_ZERO) ?
37         EXCEPTION_EXECUTE_HANDLER :
38                                     EXCEPTION_CONTINUE_SEARCH)
39     {
40         printf("External handler in action.");
41     }
42     return 0;
43 }
```

Запуск отладчика подтвердил ожидаемый результат - поиск подходящего обработчика для исключения происходит снизу вверх. При этом создаётся опасность утечки ресурсов, поэтому желательно обрабатывать исключительные ситуации в месте их возникновения.

Использование goto считается дурной практикой по целому ряду причин. В листинге 7, благодаря goto управление со строки 25 передаётся сразу на строку 32. Таким образом осуществляется выход из блока __try без возбуждения и обработки исключения.

Listing 7: 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
16 void usage(const _TCHAR *prog);
17
18 // Defines the entry point for the console application.
19 int _tmain(int argc, _TCHAR* argv[]) {
20     // Floating point exceptions are masked by default.
21     _clearfp();
22     _controlfp_s(NULL, 0, _EM_OVERFLOW | _EM_ZERODIVIDE);
23
24     __try {
25         goto OUT_POINT;
26         RaiseException(EXCEPTION_FLT_DIVIDE_BY_ZERO,
27                                             EXCEPTION_NONCONTINUABLE, 0, NULL);
28     }
29     __except (EXCEPTION_EXECUTE_HANDLER)
30     {
31         printf("Handler in action.");
32     }
33 OUT_POINT:
34     printf("A point outside the __try block.");
35     return 0;
36 }
```

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

Листинг 8 похож на листинг 7, но за пределы охраняемого фрейма кода помогает выйти на этот раз __leave. По сути, результат прежний, но этот способ считается более правильным, т.к. не приводит к раскрутке стека.

Listing 8: 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 <excpt.h>
14 #include <windows.h>
15
16 void usage(const _TCHAR *prog);
17
18 // Defines the entry point for the console application.
19 int _tmain(int argc, _TCHAR* argv[]) {
20     // Floating point exceptions are masked by default.
21     _clearfp();
22     _controlfp_s(NULL, 0, _EM_OVERFLOW | _EM_ZERODIVIDE);
23
24     __try {
25         __leave;
26         RaiseException(EXCEPTION_FLT_DIVIDE_BY_ZERO,
27                                                 EXCEPTION_NONCONTINUABLE, 0, NULL);
28     }
29     __except (EXCEPTION_EXECUTE_HANDLER)
30     {
31         printf("Handler in action.");
32     }
33
34     printf("A point outside the __try block.");
35     return 0;
36 }
```

Результат использования __leave — переход в конец блока try-finally. После перехода выполняется обработчик завершения. Хотя для получения того же результата можно использовать оператор goto, он (оператор goto) приводит к освобождению стека. Оператор __leave более эффективен, поскольку не вызывает освобождение стека.

Листинг 9 показывает встраивание SEH в механизм исключений C/C++. Для этого необходимо включить соответствующие опции в компиляторе (/EHa).

Listing 9: 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 <excpt.h>
20 #include <windows.h>
21
22 void translator(unsigned int u, EXCEPTION_POINTERS* pExp);
23
24 // Defines the entry point for the console application.
25 int _tmain(int argc, _TCHAR* argv[]) {
26     // Floating point exceptions are masked by default.
27     _clearfp();
28     _controlfp_s(NULL, 0, _EM_OVERFLOW | _EM_ZERODIVIDE);
29
30     try {
31         _set_se_translator(translator);
32         RaiseException(EXCEPTION_FLT_DIVIDE_BY_ZERO,
33                                                 EXCEPTION_NONCONTINUABLE, 0, NULL);
34     }
35     catch (std::overflow_error e) {
36         printf("Error: %s", e.what());
37     }
38     return 0;
39 }
40
41 void translator(unsigned int u, EXCEPTION_POINTERS* pExp) {
42     if (u == EXCEPTION_FLT_DIVIDE_BY_ZERO)
43         throw std::overflow_error("EXCEPTION_FLT_DIVIDE_BY_ZERO");
44 }
```

Если проследить за передачей управления по стеку вызовов, то сразу после возбуждения исключения в 32-й строке, управление передается транслятору, и только после этого в блок catch, где происходит обработка исключения. Этот механизм способен обеспечить взаимодействие SEH с другими языками и системами.

В листинге 10 исключение как таковое отсутствует, но есть охраняемый блок кода, и блок __finally, управление в который будет передано в любой ситуации.

Listing 10: 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
14 void usage(const _TCHAR *prog);
15
16 // Defines the entry point for the console application.
17 int _tmain(int argc, _TCHAR* argv[]) {
18     // Floating point exceptions are masked by default.
19     _clearfp();
20     _controlfp_s(NULL, 0, _EM_OVERFLOW | _EM_ZERODIVIDE);
21
22     __try {
23         // No exception
24     }
25     __finally
26     {
27         printf("Thre is no exception, but the handler is called.\n");
28     }
29
30     return 0;
31 }
```

Вместо передачи управления обратно в программу, управление передаётся в блок __finally. Похожие механизмы есть в других распространённых языках программирования, они позволяют обеспечить строгие гарантии исключений, и не допустить нахождение объекта в не консестентном состоянии.

В листинге 11 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.

Listing 11: 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
17 void usage(const _TCHAR *prog);
18
19 // Defines the entry point for the console application.
20 int _tmain(int argc, _TCHAR* argv[]) {
21     // Floating point exceptions are masked by default.
22     _clearfp();
23     _controlfp_s(NULL, 0, _EM_OVERFLOW | _EM_ZERODIVIDE);
24
25     __try {
26         goto OUT_POINT;
27         RaiseException(EXCEPTION_FLT_DIVIDE_BY_ZERO,
28                                                 EXCEPTION_NONCONTINUABLE, 0, NULL);
29     }
30     __finally
31     {
32         if (AbnormalTermination())
33             printf("%s", "Abnormal termination in goto case\n");
34         else
35             printf("%s", "Normal termination  in goto case\n");
36     }
37 OUT_POINT:
38     __try {
39         __leave;
40         RaiseException(EXCEPTION_FLT_DIVIDE_BY_ZERO,
41                                                 EXCEPTION_NONCONTINUABLE, 0, NULL);
42     }
43     __finally
44     {
45         if (AbnormalTermination())
46             printf("%s", "Abnormal termination in __leave case");
47         else
48             printf("%s", "Normal termination  in __leave case");
49     }
50
```

```
51      return 0;
52  }
```

Функция AbnormalTermination() не в состоянии отследить выход по goto и всё время возвращает результат правильно завершения (как при возбуждении исключения, так и до этого), но она правильно отрабатывает при вызове __leave (вероятно, благодаря тому, что __leave выставляет нужные флаги). Следовательно, 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.

Вывод

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 уровня языка C++ это разные инструменты, требующие разного подхода.