Санкт-Петербургский Политехнический Университет Петра Великого

Кафедра компьютерных систем и программных технологий

Отчёт по лабораторной работе №1

Курс: «Системное программирование»

Тема: «Обработка исключений в ОС Windows»

Выполнил студент:

Волкова Мария Дмитриевна
Группа: 13541/2

Проверил:

Душутина Елена Владимировна

Санкт-Петербург
2018 г.

Содержание

Лабораторная работа №1

1.1 Цель работы

Научиться обрабатывать исключения с помощью встроенных средств WinAPI. Использовать системные утилиты для получения информации о системной активности процесса.

1.2 Программа работы

1. Сгенерировать и обработать исключения с помощью функций WinAPI;
2. Получить код исключения с помощью функции GetExceptionCode.
   - Использовать эту функции в выражении фильтре;
   - Использовать эту функцию в обработчике.
3. Создать собственную функцию-фильтр;
4. Получить информацию об исключении с помощью функции GetExceptionInformation; сгенерировать исключение с помощью функции RaiseException;
5. Использовать функции UnhandleExceptionFilter и Set UnhandleExceptionFilter для необработанных исключений;
6. Обработать вложенные исключения;
7. Выйти из блока __try с помощью оператора goto;
8. Выйти из блока __try с помощью оператора leave;
9. Преобразовать структурное исключение в исключение языка C, используя функцию translator;
10. Использовать финальный обработчик finally;
11. Проверить корректность выхода из блока __try с помощью функции AbnormalTermination в финальном обработчике finally.

1.3 Характеристики системы

Некоторая информация об операционной системе и ресурсах системы:

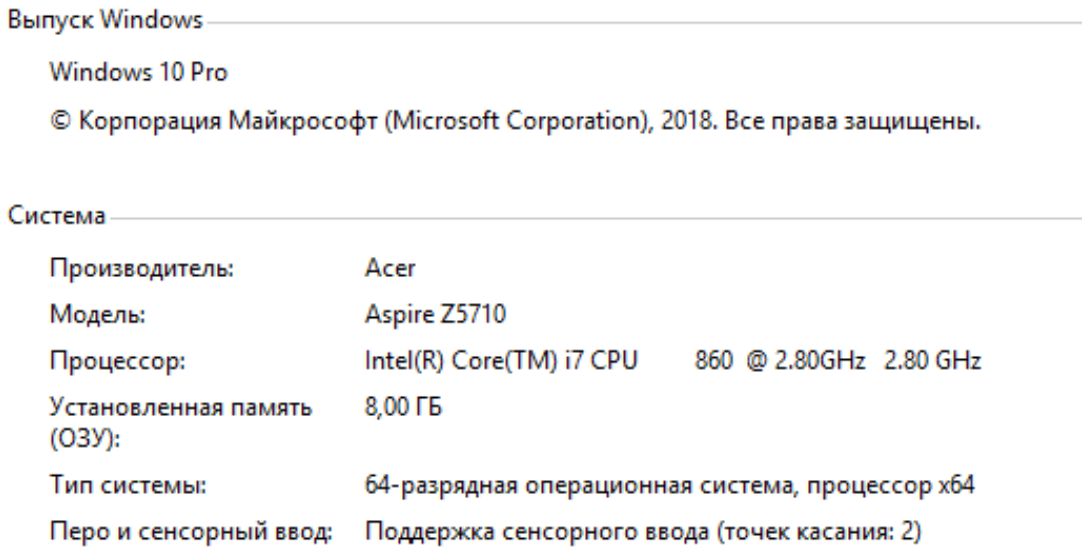

Рис. 1.1: Конфигурация системы

Информация о компиляторе (был использован компилятор Microsoft и GCC):

```
Оптимизирующий
 компилятор Microsoft (R) C/C++ версии 19.15.26732.1 для x64
(C) Корпорация Майкрософт (Microsoft Corporation). Все права защищены.
```

1.4 Ход работы

1.4.1 Генерация и обработка исключения средствами WinAPI

В операционной системе Windows используется механизм структурной обработки исключений (SEH). В отличие от встроенных средств обработки исключений языка C++, SEH позволяет обрабатывать не только программные исключения, но и аппаратные.

```
__try {
  // Защищенный код.
} __except ( /* Фильтр исключений. */ ) {
  // Обработчик исключений.
}
```

Если при выполнении защищенного кода из блока __try возникнет исключение, то операционная система перехватит его и приступит к поиску блока __except. Найдя его, она передаст управление фильтру исключений. Фильтр исключений может получить код исключения и на основе этого кода принять решение, передать управление обработчику или же сказать системе, чтобы она искала предыдущий по вложенности блок __except [1] Фильтр исключений может возвращать одно из трех значений (идентификаторов), которые определены в файле excpt.h:

- Идентификатор EXCEPTION_EXECUTE_HANDLER означает, что для этого блока __try есть обработчик исключения и он готов обработать это исключение.
- Идентификатор EXCEPTION_CONTINUE_SEARCH означает, что для обработки исключения существует предыдущий по вложенности блок __except.
- Идентификатор EXCEPTION_CONTINUE_EXECUTION означает, что выполнение продолжится с того участка кода, который вызвал исключение.

Подобная система обработки исключений позволяет организовывать вложенные исключения, что значительно увеличивает гибкость и читабельность языка программирования.

Некоторые типы исключений, которые могут быть обработаны в фильтре [2]:

- EXCEPTION_ACCESS_VIOLATION – попытка чтения или записи в виртуальную память без соответствующих прав доступа;
- EXCEPTION_BREAKPOINT – встретилась точка останова;
- EXCEPTION_DATATYPE_MISALIGNMENT – доступ к данным, адрес которых не выровнен по границе слова или двойного слова;
- EXCEPTION_SINGLE_STEP – механизм трассировки программы сообщает, что выполнена одна инструкция;
- EXCEPTION_ARRAY_BOUNDS_EXCEEDED – выход за пределы массива, если аппаратное обеспечение поддерживает такую проверку;
- EXCEPTION_FLT_DENORMAL_OPERAND – один из операндов с плавающей точкой является ненормализованным;
- EXCEPTION_FLT_DIVIDE_BY_ZERO – попытка деления на ноль в операции с плавающей точкой;
- EXCEPTION_FLT_INEXACT_RESULT – результат операции с плавающей точкой не может быть точно представлен десятичной дробью;
- EXCEPTION_FLT_INVALID_OPERATION – ошибка в операции с плавающей точкой, для которой не предусмотрены другие коды исключения;
- EXCEPTION_FLT_OVERFLOW – при выполнении операции с плавающей точкой произошло переполнение;
- EXCEPTION_FLT_STACK_CHECK – переполнение или выход за нижнюю границу стека при выполнении операции с плавающей точкой;
- EXCEPTION_FLT_UNDERFLOW – результат операции с плавающей точкой является числом, которое меньше минимально возможного числа с плавающей точкой;

- EXCEPTION_INT_DIVIDE_BY_ZERO – попытка деления на ноль при операции с целыми числами;
- EXCEPTION_INT_OVERFLOW – при выполнении операции с целыми числами произошло переполнение;
- EXCEPTION_PRIV_INSTRUCTION – попытка выполнения привилегированной инструкции процессора, которая недопустима в текущем режиме процессора;
- EXCEPTION_NONCONTINUABLE_EXCEPTION – попытка возобновления исполнения программы после исключения, которое запрещает выполнять такое действие.

Разработаем программу, которая генерирует исключение EXCEPTION_INT_DIVIDE_BY_ZERO из защищенного участка кода и выводит его шестнадцатиричный код в обработчике:

```
include <iostream>
#include <windows.h>

int main() {
  __try {
    RaiseException(EXCEPTION_INT_DIVIDE_BY_ZERO, NULL, NULL, nullptr);
  } __except(GetExceptionCode() == EXCEPTION_INT_DIVIDE_BY_ZERO ?
    EXCEPTION_EXECUTE_HANDLER : EXCEPTION_CONTINUE_SEARCH) {
    std::cerr << "EXCEPTION_INT_DIVIDE_BY_ZERO: 0x" << std::hex <<
    EXCEPTION_INT_DIVIDE_BY_ZERO << std::endl;
    return 0x1;
  }

  return 0x0;
}
```

Результат работы программы:

```
EXCEPTION_INT_DIVIDE_BY_ZERO: 0xc0000094

C:\SP\Debug\SP.exeпроцесс( 12184) завершает работу с кодом 1.
```

Попробуем обработать аппаратное исключение EXCEPTION_NONCONTINUABLE_EXCEPTION:

```
#include <iostream>
#include <windows.h>

int main() {
  __try {

    RaiseException(
      1,                    // exception code
      EXCEPTION_NONCONTINUABLE,                    // continuable exception
      0, NULL);             // no arguments
  } __except ((GetExceptionCode() == EXCEPTION_NONCONTINUABLE_EXCEPTION)
    ? EXCEPTION_EXECUTE_HANDLER : EXCEPTION_CONTINUE_EXECUTION){
    std::cerr << "EXCEPTION_NONCONTINUABLE_EXCEPTION: 0x" << std::hex <<
    EXCEPTION_NONCONTINUABLE_EXCEPTION << std::endl;
    return 0x1;
  }

  return 0x0;
}
```

Результат работы программы:

```
EXCEPTION_NONCONTINUABLE_EXCEPTION: 0xc0000025

C:\SP\Debug\SP.exe процесс( 10132) завершает работу с кодом 1.
```

Оба исключения были успешно обработаны. В качестве фильтра для обработчика был использован простой тернарный оператор, который вызывает обработчик только при возникновении конкретного исключения. Если исключение не было вызвано, то поиск обработчика продолжится, что для данного случая эквивалентно падению программы.

Первой выполняемой инструкцией данной программы является обращение к библиотеке ядра Windows KERNEL32.DLL функцией RaiseException. В результате выбрасывается исключение с кодом, которое не совпадает с кодом 0xc0000025 (EXCEPTION_NONCONTINUABLE_EXCEPTION), поэтому выполняется инструкция EXCEPTION_CONTINUE_EXECUTION, но исключение было помечено как NONCONTINUABLE, поэтому вызывается EXCEPTION_NONCONTINUABLE_EXCEPTION. Все последующие операции производятся уже в обработчике исключения (_except).

1.4.2 Получить код исключения с помощью функции GetExceptionCode

Использовать эту функцию в выражении фильтре

Для получения кода выброшенного исключения, вызывается функция GetExceptionCode, имеющая следующий прототип:

```
DWORD GetExceptionCode(void);
```

Функцию можно вызывать внутри фильтра, а также непосредственно внутри обработчика исключения. Правилом хорошего тона при программировании является явное указание в фильтре всех возможных для данного охраняемого кода исключений. Таким образом те исключения, о которых программист не подумал дадут о себе знать на этапе отладки.

Использовать эту функцию в обработчике

Использование функции GetExceptionCode возможно также непосредственно внутри обработчика, однако, не стоит подменять фильтр этой возможностью, особенно при работе со вложенными исключениями.

Разработаем программу, получающую код исключения непосредственно внутри обработчика:

```
#include <iostream>
#include <windows.h>

int main() {
  auto zero = 0, one = 1;
  __try {
    const auto number = one / zero;
  } __except(EXCEPTION_EXECUTE_HANDLER) {
    const auto exceptionCode = GetExceptionCode();
    std::cerr << "EXCEPTION_CODE: 0x" << std::hex << exceptionCode << std::endl;
    return 0x1;
  }

  return 0x0;
}
```

Результат работы программы:

```
EXCEPTION_CODE: 0xc0000094

C:\SP\Debug\SP.exe процесс( 2172) завершает работу с кодом 1.
```

Теперь разработаем аналогичную программу для исключения при доступе к данным с невыровненным адресом:

```
#include <iostream>
#include <windows.h>


int main() {

  __try {
    RaiseException(EXCEPTION_NONCONTINUABLE_EXCEPTION, NULL, NULL, nullptr);
  } __except(EXCEPTION_EXECUTE_HANDLER) {
    const auto exceptionCode = GetExceptionCode();
    std::cerr << "EXCEPTION_CODE: 0x" << std::hex << exceptionCode << std::endl;
    return 0x1;
  }

```

```
15    return 0x0;
16 }
```

Результат работы программы:

```
1 EXCEPTION_CODE: 0xc0000025
2
3 C:\SP\Debug\SP.exe процесс( 22956) завершает работу с кодом 1.
```

В результате эксперимента было выявлено, что функция GetExceptionCode выдает корректный результат при вызове непосредственно внутри обработчика.

1.4.3 Создать собственную функцию-фильтр

Для больших программ, в которых использование фильтрующего тернарного оператора недостаточно, функциональность фильтра бывает удобно перенести во внешнюю функцию. При этом данная функция должна возвращать один из трех идентификаторов, определяющих поведение программы.

Разработаем программу с отдельной фильтрующей функцией:

```
1 #include <iostream>
2 #include <windows.h>
3
4
5 LONG filter(DWORD actual_code, DWORD expected_code) {
6   if (actual_code == expected_code) {
7     return EXCEPTION_EXECUTE_HANDLER;
8   }
9   else {
10    return EXCEPTION_CONTINUE_SEARCH;
11  }
12
13 }
14
15 int main()
16 {
17   __try
18   {
19     RaiseException(EXCEPTION_INT_DIVIDE_BY_ZERO, NULL, NULL, nullptr);
20   }
21   __except (filter(GetExceptionCode(), EXCEPTION_INT_DIVIDE_BY_ZERO))
22   {
23     std::cout << "Caught EXCEPTION_INT_DIVIDE_BY_ZERO" << std::endl;
24   }
25   __try
26   {
27     RaiseException(EXCEPTION_NONCONTINUABLE_EXCEPTION, NULL, NULL, nullptr);
28   }
29   __except (filter(GetExceptionCode(), EXCEPTION_NONCONTINUABLE_EXCEPTION))
30   {
31     std::cout << "Caught EXCEPTION_NONCONTINUABLE_EXCEPTION" << std::endl;
32   }
33   return 0;
34 }
```

Результат работы программы:

```
1 Caught EXCEPTION_INT_DIVIDE_BY_ZERO
2 Caught EXCEPTION_NONCONTINUABLE_EXCEPTION
3
4 C:\SP\Debug\SP.exe процесс( 7420) завершает работу с кодом 0.
```

1.4.4 Получить информацию об исключении, сгенерировать исключение

Помимо функции GetExceptionCode внутри фильтра или обработчика можно вызвать функцию GetExceptionInfo которая возвращает заполненную структуру, в полях которой содержится детальная информация об исключении. Сигнатура функции выглядит следующим образом:

```
LPEXCEPTION_POINTERS GetExceptionInformation(void);
```

Возвращаемая структура содержит указатели на структуру EXCEPTION_RECORD и на структуру CONTEXT. Структура EXCEPTION_RECORD содержит машинно-независимое описание исключения, CONTEXT содержит специфическое состояние процессора на момент исключения [4]

В большинстве случаев интересующая программиста информация содержится в структуре EXCEPTION_RECO со следующими полями:

- ExceptionCode – код исключения, совпадающий с результатом функции GetExceptionCode.
- ExceptionFlags – флаги исключения.
- ExceptionAddress – адрес, по которому было вызвано исключение.
- NumberParameters – количество элементов массива ExceptionInformation.
- ExceptionInformation – массив, содержащий наиболее детальную информацию о исключении. Может быть задан функцией RaiseException, а также определен для специфических исключений (например EXCEPTION_ACCESS_VIOLATION). В остальных случаях массив не определен.
- ExceptionRecord – указатель на связанную структуру EXCEPTION_RECORD.

Исключение можно сгенерировать функцией RaiseException со следующей сигнатурой [5]:

```
void WINAPI RaiseException(
    _In_       DWORD      dwExceptionCode,
    _In_       DWORD      dwExceptionFlags,
    _In_       DWORD      nNumberOfArguments,
    _In_ const ULONG_PTR *lpArguments
);
```

В качестве параметров обязательно указывается код исключения, который может быть собственным. Необязательными параметрами являются флаги исключения, а также массив детализированной информации с указанием количества аргументов.

Разработаем фильтрующую функцию, которая записывает в глобальную переменную информацию о последнем возникшем исключении, после чего в обработчике использует эти данные, если необходимо:

```
#include <iostream>
#include <windows.h>

EXCEPTION_RECORD exceptionRecord;

DWORD filterException(const DWORD exceptionCode, const EXCEPTION_POINTERS*
    exceptionPointers);

int main() {
  __try {
    RaiseException(EXCEPTION_INT_DIVIDE_BY_ZERO, NULL, NULL, nullptr);
  } __except(filterException(GetExceptionCode(), GetExceptionInformation())) {
    std::cerr << "EXCEPTION_INT_DIVIDE_BY_ZERO: 0x" << std::hex <<
    EXCEPTION_INT_DIVIDE_BY_ZERO << std::endl;
    std::cout << "Exception code: 0x" << std::hex << exceptionRecord.ExceptionCode << std
    ::endl;
    std::cout << "Exception flags: 0x" << std::hex << exceptionRecord.ExceptionFlags <<
    std::endl;
    std::cout << "Exception record: 0x" << std::hex << exceptionRecord.ExceptionRecord <<
     std::endl;
    std::cout << "Exception address: 0x" << std::hex << exceptionRecord.ExceptionAddress
    << std::endl;
    std::cout << "Number parameters: " << exceptionRecord.NumberParameters << std::endl;
    return 0x1;
  }

  return 0x0;
}

```

```
24 DWORD filterException(const DWORD exceptionCode, const EXCEPTION_POINTERS*
       exceptionPointers) {
25   memcpy(&exceptionRecord, exceptionPointers->ExceptionRecord, sizeof(exceptionRecord));
26   return exceptionCode == EXCEPTION_INT_DIVIDE_BY_ZERO ? EXCEPTION_EXECUTE_HANDLER :
       EXCEPTION_CONTINUE_SEARCH;
27 }
```

Результат работы программы:

```
1 EXCEPTION_INT_DIVIDE_BY_ZERO: 0xc0000094
2 Exception code: 0xc0000094
3 Exception flags: 0x0
4 Exception record: 0x00000000
5 Exception address: 0x77991812
6 Number parameters: 0
7
8 C:\SP\Debug\SP.exe процесс( 23300) завершает работу с кодом 1.
```

Разработаем аналогичную программу для исключения EXCEPTION_NONCONTINUABLE_EXCEPTION:

```
1  #include <iostream>
2  #include <windows.h>
3
4  EXCEPTION_RECORD exceptionRecord;
5
6  DWORD filterException(const DWORD exceptionCode, const EXCEPTION_POINTERS*
       exceptionPointers);
7
8  int main() {
9    __try {
10     RaiseException(EXCEPTION_NONCONTINUABLE_EXCEPTION, NULL, NULL, nullptr);
11   }
12   __except (filterException(GetExceptionCode(), GetExceptionInformation())) {
13     std::cerr << "EXCEPTION_NONCONTINUABLE_EXCEPTION: 0x" << std::hex <<
       EXCEPTION_NONCONTINUABLE_EXCEPTION << std::endl;
14     std::cout << "Exception code: 0x" << std::hex << exceptionRecord.ExceptionCode << std
       ::endl;
15     std::cout << "Exception flags: 0x" << std::hex << exceptionRecord.ExceptionFlags <<
       std::endl;
16     std::cout << "Exception record: 0x" << std::hex << exceptionRecord.ExceptionRecord <<
        std::endl;
17     std::cout << "Exception address: 0x" << std::hex << exceptionRecord.ExceptionAddress
       << std::endl;
18     std::cout << "Number parameters: " << exceptionRecord.NumberParameters << std::endl;
19     return 0x1;
20   }
21
22   return 0x0;
23 }
24
25 DWORD filterException(const DWORD exceptionCode, const EXCEPTION_POINTERS*
       exceptionPointers) {
26   memcpy(&exceptionRecord, exceptionPointers->ExceptionRecord, sizeof(exceptionRecord));
27   return exceptionCode == EXCEPTION_NONCONTINUABLE_EXCEPTION ? EXCEPTION_EXECUTE_HANDLER
       : EXCEPTION_CONTINUE_SEARCH;
28 }
```

Результат работы программы:

```
1 EXCEPTION_NONCONTINUABLE_EXCEPTION: 0xc0000025
2 Exception code: 0xc0000025
3 Exception flags: 0x0
4 Exception record: 0x00000000
5 Exception address: 0x77991812
6 Number parameters: 0
7
8 C:\SP\Debug\SP.exe процесс( 4484) завершает работу с кодом 1.
```

1.4.5 Использовать функции UnhandleExceptionFilter и SetUnhandleExceptionFilter для необработанных исключений

Помимо конструкции __try и __except в Windows имеется возможность превратить весь код в охраняемый, посредством установления фильтра на необработанные исключения. По большей части это считается плохим тоном, однако может быть полезно на стадии релиза приложения, когда при возникновении необработанной ошибки программа должна показывать диалоговое окно с ее описанием.

Установление фильтра на необработанные исключения производится функцией SetUnhandleExceptionFilter, которая имеет следующую сигнатуру [6]:

```
LPTOP_LEVEL_EXCEPTION_FILTER WINAPI SetUnhandledExceptionFilter(
    _In_ LPTOP_LEVEL_EXCEPTION_FILTER lpTopLevelExceptionFilter
);
```

В качестве единственного аргумента принимается указатель на функцию фильтр. Возвращаемое значение содержит указатель на предыдущую функцию обработчик.

Разработаем программу, которая устанавливает фильтр необработанных исключений, выводит адрес старого фильтра, нового фильтра и после этого выбрасывает исключение.

```
#include <iostream>
#include <windows.h>

EXCEPTION_RECORD exceptionRecord;

LONG WINAPI exceptionFilter(EXCEPTION_POINTERS* exceptionInformation);

int main() {
  // Устанавливаем обработчик исключений для всех необработанных исключений.
  const auto filterPointer = SetUnhandledExceptionFilter(exceptionFilter);
  std::cout << "OLD_EXCEPTION_HANDLER: 0x" << std::hex << filterPointer << std::endl
          << "NEW_EXCEPTION_HANDLER: 0x" << std::hex << exceptionFilter << std::endl;

  RaiseException(EXCEPTION_NONCONTINUABLE_EXCEPTION, NULL, NULL, nullptr);
  return 0x0;
}

LONG WINAPI exceptionFilter(EXCEPTION_POINTERS* exceptionInformation) {
  std::cout << "EXCEPTION_CODE: 0x" << std::hex << exceptionInformation->ExceptionRecord
    ->ExceptionCode << std::endl;
  return EXCEPTION_EXECUTE_HANDLER;
}
```

Результат работы программы:

```
OLD_EXCEPTION_HANDLER: 0x010211B8
NEW_EXCEPTION_HANDLER: 0x01021186
EXCEPTION_CODE: 0xc0000025

C:\SP\Debug\SP.exe процесс( 5363) завершает работу с кодом 0.
```

1.4.6 Обработать вложенные исключения

Архитектура обработки исключений в WinAPI позволяет обрабатывать вложенные блоки __try, __except. Для того, чтобы передать управление внешнему обработчику исключений из внутреннего, фильтр внутреннего обработчика должен возвращать EXCEPTION_CONTINUE_SEARCH.

Разработаем программу, которая содержит один блок __try, __except внутри другого. Внутренний обработчик вызывается на исключение EXCEPTION_INT_DIVIDE_BY_ZERO, в противном случае продолжит искать обработчик дальше. Внешний обработчик вызывается на исключение EXCEPTION_NONCONTINUABLE_E
в противном случае продолжит искать обработчик дальше. Внутри охраняемого кода вызывается соответствующее исключение, в зависимости от аргумента командной строки.

```
#include <iostream>
#include <windows.h>

```

```
4  DWORD filterException(const DWORD exceptionCode, const DWORD exceptionExpect);
5
6  int main(const int argc, const char** argv) {
7    __try {
8      __try {
9        // Если первый аргумент равен '1', то вызываем исключение
       EXCEPTION_NONCONTINUABLE_EXCEPTION, если нет, то EXCEPTION_INT_DIVIDE_BY_ZERO.
10       const auto exceptionCode = (argc > 1 && std::strcmp(argv[1], "1") == 0) ?
       EXCEPTION_NONCONTINUABLE_EXCEPTION : EXCEPTION_INT_DIVIDE_BY_ZERO;
11       RaiseException(exceptionCode, NULL, NULL, nullptr);
12     }
13     __except(filterException(GetExceptionCode(), EXCEPTION_INT_DIVIDE_BY_ZERO)) {
14       std::cout << "EXCEPTION_INT_DIVIDE_BY_ZERO" << std::endl;
15       return 0x1;
16     }
17   } __except(filterException(GetExceptionCode(), EXCEPTION_NONCONTINUABLE_EXCEPTION)) {
18     std::cout << "EXCEPTION_NONCONTINUABLE_EXCEPTION" << std::endl;
19     return 0x2;
20   }
21
22   return 0x0;
23 }
24
25 DWORD filterException(const DWORD exceptionCode, const DWORD exceptionExpect) {
26   if(exceptionCode == exceptionExpect) {
27     // Вызываем обработчик.
28     std::cout << "EXCEPTION_EXECUTE_HANDLER" << std::endl;
29     return EXCEPTION_EXECUTE_HANDLER;
30   }
31
32   // Продолжаем поиск обработчика выше.
33   std::cout << "EXCEPTION_CONTINUE_SEARCH" << std::endl;
34   return EXCEPTION_CONTINUE_SEARCH;
35 }
```

Результат работы программы:

```
1  C
2  -- аргументом 1
3  EXCEPTION_CONTINUE_SEARCH
4  EXCEPTION_EXECUTE_HANDLER
5  EXCEPTION_NONCONTINUABLE_EXCEPTIONC
6
7  -- аргументом 2
8  EXCEPTION_EXECUTE_HANDLER
9  EXCEPTION_INT_DIVIDE_BY_ZERO
```

В первом варианте (аргумент 0 – EXCEPTION_INT_DIVIDE_BY_ZERO) внутренний фильтр вернул EXCEPTION_EXECUTE_HANDLER после чего был вызван обработчик и приложение завершилось.

Во втором варианте (аргумент 1 – EXCEPTION_NONCONTINUABLE_EXCEPTION) внутренний фильтр вернул EXCEPTION_CONTINUE_SEARCH после чего был найден внешний обработчик. Внешний фильтр вернул EXCEPTION_EXECUTE_HANDLER, был вызван обработчик и приложение завершилось.

1.4.7 Выйти из блока __try с помощью оператора goto

В WinAPI, а также в стандарте языка C++ разрешается выходить из охраняемого кода при помощи оператора goto. Использование оператора goto является плохой практикой в современных языках программирования из-за своей неэффективности. Каждый раз при использовании goto вызывается раскрутка стека, что делает данную операцию весьма медленной.

Разработаем программу, которая в зависимости от аргумента командной строки, выходит из охраняемого кода через goto или __leave:

```
1  #include <iostream>
2  #include <windows.h>
3
4  int main(const int argc, const char** argv) {
```

```
5     __try {
6
7       // Если первый аргумент равен '1', то выходим через goto, если нет, то через __leave.
8       const auto isGoto = argc > 1 && std::strcmp(argv[1], "1") == 0;
9       if(isGoto)
10        goto out;
11      else
12        __leave;
13
14      // Программа должна завершиться до вызова исключения.
15      RaiseException(EXCEPTION_NONCONTINUABLE_EXCEPTION, NULL, NULL, nullptr);
16    } __except(EXCEPTION_EXECUTE_HANDLER) {
17      std::cout << "Exception!" << std::endl;
18      return 0x1;
19    }
20
21    std::cout << "Leave with __leave" << std::endl;
22    return 0x0;
23
24  out:
25    std::cout << "Leave with goto" << std::endl;
26    return 0x0;
27  }
```

Результат работы программы:

```
1  C
2  -- аргументом 1
3  Leave with gotoC
4
5  -- аргументом 2
6  Leave with __leave
```

Таким образом, goto позволил выйти из охраняемого кода. В отличие от goto, команда __leave позволяет выходить только из охраняемого кода. Использование __leave в другом месте вызовет ошибку компиляции. Кроме того, __leave не позволяет выходить из вложенных блоков __try, __except. Однако, использование этой команды рекомендуется при работе с исключениями, потому что не вызывает раскрутку стека, а также увеличивает читабельность кода

Скомпилированный ассемблерный код:

```
1  #include <iostream>
2  #include <windows.h>
3
4  int main(const int argc, const char** argv) {
5  000725B0  push        ebp
6  000725B1  mov         ebp,esp
7  000725B3  push        0FFFFFFFEh
8  000725B5  push        7AF50h
9  000725BA  push        offset _except_handler4 (072B30h)
10 000725BF  mov         eax,dword ptr fs:[00000000h]
11 000725C5  push        eax
12 000725C6  add         esp,0FFFFFF1Ch
13 000725CC  push        ebx
14 000725CD  push        esi
15 000725CE  push        edi
16 000725CF  lea         edi,[ebp-0F4h]
17 000725D5  mov         ecx,37h
18 000725DA  mov         eax,0CCCCCCCCh
19 000725DF  rep stos     dword ptr es:[edi]
20 000725E1  mov         eax,dword ptr [__security_cookie (07C004h)]
21 000725E6  xor         dword ptr [ebp-8],eax
22 000725E9  xor         eax,ebp
23 000725EB  push        eax
24 000725EC  lea         eax,[ebp-10h]
25 000725EF  mov         dword ptr fs:[00000000h],eax
26 000725F5  mov         dword ptr [ebp-18h],esp
27 000725F8  mov         ecx,offset _08935D38_7@cpp (07F033h)
```

```
000725FD  call        @__CheckForDebuggerJustMyCode@4 (071276h)
  __try {
00072602  mov         dword ptr [ebp-4],0

    // Если первый аргумент равен '1', то выходим через goto, если нет, то через __leave.
    const auto isGoto = argc > 1 && std::strcmp(argv[1], "1") == 0;
00072609  cmp         dword ptr [argc],1

    // Если первый аргумент равен '1', то выходим через goto, если нет, то через __leave.
    const auto isGoto = argc > 1 && std::strcmp(argv[1], "1") == 0;
0007260D  jle         main+88h (072638h)
0007260F  push        offset string "1" (079B30h)
00072614  mov         eax,4
00072619  shl         eax,0
0007261C  mov         ecx,dword ptr [argv]
0007261F  mov         edx,dword ptr [ecx+eax]
00072622  push        edx
00072623  call        _strcmp (07129Eh)
00072628  add         esp,8
0007262B  test        eax,eax
0007262D  jne         main+88h (072638h)
0007262F  mov         byte ptr [ebp-0F1h],1
00072636  jmp         main+8Fh (07263Fh)
00072638  mov         byte ptr [ebp-0F1h],0
0007263F  mov         al,byte ptr [ebp-0F1h]
00072645  mov         byte ptr [ebp-1Dh],al
    if(isGoto)
00072648  movzx       eax,byte ptr [ebp-1Dh]
0007264C  test        eax,eax
0007264E  je          main+0AEh (07265Eh)
      goto out;
00072650  mov         dword ptr [ebp-4],0FFFFFFFEh
00072657  jmp         $LN11+79h (072702h)
    else
0007265C  jmp         main+0B0h (072660h)
      __leave;
0007265E  jmp         main+0CAh (07267Ah)

    // Программа должна завершиться до вызова исключения.
    RaiseException(EXCEPTION_NONCONTINUABLE_EXCEPTION, NULL, NULL, nullptr);
00072660  mov         esi,esp
00072662  push        0
00072664  push        0
00072666  push        0
00072668  push        0C0000025h
0007266D  call        dword ptr [__imp__RaiseException@16 (07D000h)]

    // Программа должна завершиться до вызова исключения.
    RaiseException(EXCEPTION_NONCONTINUABLE_EXCEPTION, NULL, NULL, nullptr);
00072673  cmp         esi,esp
00072675  call        __RTC_CheckEsp (071280h)
  } __except(EXCEPTION_EXECUTE_HANDLER) {
0007267A  mov         dword ptr [ebp-4],0FFFFFFFEh
00072681  jmp         $LN11+4Ch (0726D5h)
$LN15:
00072683  mov         eax,1
$LN14:
00072688  ret
$LN11:
00072689  mov         esp,dword ptr [ebp-18h]
    std::cout << "Exception!" << std::endl;
0007268C  mov         esi,esp
0007268E  push        offset std::endl<char,std::char_traits<char>> (0712A8h)
00072693  push        offset string "Exception!" (079B34h)
00072698  mov         eax,dword ptr [_imp_?cout@std@@3V?$basic_ostream@DU?
    $char_traits@D@std@@@1@A (07D098h)]
```

```
93  0007269D  push        eax
94  0007269E  call        std::operator<<<std::char_traits<char> > (071208h)
95  000726A3  add         esp,8
96  000726A6  mov         ecx,eax
97  000726A8  call        dword ptr [__imp_std::basic_ostream<char,std::char_traits<char> >::
        operator<< (07D0D4h)]
98  000726AE  cmp         esi,esp
99  000726B0  call        __RTC_CheckEsp (071280h)
100     return 0x1;
101 000726B5  mov         dword ptr [ebp-0ECh],1
102 000726BF  mov         dword ptr [ebp-4],0FFFFFFFEh
103 000726C6  mov         eax,dword ptr [ebp-0ECh]
104 000726CC  jmp         $LN11+0A4h (07272Dh)
105   } __except(EXCEPTION_EXECUTE_HANDLER) {
106 000726CE  mov         dword ptr [ebp-4],0FFFFFFFEh
107   }
108
109   std::cout << "Leave with __leave" << std::endl;
110 000726D5  mov         esi,esp
111 000726D7  push        offset std::endl<char,std::char_traits<char> > (0712A8h)
112 000726DC  push        offset string "Leave with __leave" (079B44h)
113 000726E1  mov         eax,dword ptr [_imp_?cout@std@@3V?$basic_ostream@DU?
        $char_traits@D@std@@@1@A (07D098h)]
114 000726E6  push        eax
115 000726E7  call        std::operator<<<std::char_traits<char> > (071208h)
116 000726EC  add         esp,8
117 000726EF  mov         ecx,eax
118 000726F1  call        dword ptr [__imp_std::basic_ostream<char,std::char_traits<char> >::
        operator<< (07D0D4h)]
119 000726F7  cmp         esi,esp
120 000726F9  call        __RTC_CheckEsp (071280h)
121   return 0x0;
122 000726FE  xor         eax,eax
123 00072700  jmp         $LN11+0A4h (07272Dh)
124
125 out:
126   std::cout << "Leave with goto" << std::endl;
127 00072702  mov         esi,esp
128 00072704  push        offset std::endl<char,std::char_traits<char> > (0712A8h)
129 00072709  push        offset string "Leave with goto" (079B5Ch)
130 0007270E  mov         eax,dword ptr [_imp_?cout@std@@3V?$basic_ostream@DU?
        $char_traits@D@std@@@1@A (07D098h)]
131 00072713  push        eax
132 00072714  call        std::operator<<<std::char_traits<char> > (071208h)
133 00072719  add         esp,8
134 0007271C  mov         ecx,eax
135 0007271E  call        dword ptr [__imp_std::basic_ostream<char,std::char_traits<char> >::
        operator<< (07D0D4h)]
136 00072724  cmp         esi,esp
137 00072726  call        __RTC_CheckEsp (071280h)
138   return 0x0;
139 0007272B  xor         eax,eax
140 }
```

1.4.8 Преобразовать структурное исключение в исключение языка C, используя функцию translator

Использовать одновременно исключения обоих типов, в программе на C++ проблематично, так как придется их обрабатывать по отдельности. Что-бы этого избежать SEH исключение нужно транслировать в обычное исключение. Делается это с помощью функции _set_se_translator стандартной библиотеки, сама эта функция стандартной не является. Она получает указатель на функцию транслятор, которая получает структуру описывающую исключение и в ответ, должна бросить типизированное исключение.

Сигнатура функции _set_se_translator выглядит следующим образом:

```
_se_translator_function _set_se_translator(
    _se_translator_function seTransFunction
);
```

Функция принимает на вход функцию транслятора и не возвращает значение. Функция транслятор принимает код структурного исключения и информацию о нем. Транслятор должен выбрасывать соответствующее исключение C++. Для того, чтобы компилятор MSVC разрешил использовать _set_se_translator, обязательно использование флага компилятора /EHa [8]

Разработаем программу, которая транслирует структурное исключение, вызванное функцией RaiseException в исключение языка C++.

```
#include <iostream>
#include <windows.h>

void translator(const UINT exceptionCode, EXCEPTION_POINTERS* exceptionInformation);

int main() {
  // Работает только со включенной опцией /EHa в компиляторе VS.
  _set_se_translator(translator);

  try {
    RaiseException(EXCEPTION_NONCONTINUABLE_EXCEPTION, NULL, NULL, nullptr);
  } catch(const EXCEPTION_POINTERS* exceptionInformation) {
    const auto exceptionRecord = exceptionInformation->ExceptionRecord;
    std::cout << "Exception code: 0x" << std::hex << exceptionRecord->ExceptionCode <<
    std::endl;
    std::cout << "Exception flags: 0x" << std::hex << exceptionRecord->ExceptionFlags <<
    std::endl;
    std::cout << "Exception record: 0x" << std::hex << exceptionRecord->ExceptionRecord
    << std::endl;
    std::cout << "Exception address: 0x" << std::hex << exceptionRecord->ExceptionAddress
    << std::endl;
    std::cout << "Number parameters: " << exceptionRecord->NumberParameters << std::endl;
    return 0x2;
  }

  return 0x0;
}

void translator(const UINT exceptionCode, EXCEPTION_POINTERS* exceptionInformation) {
  EXCEPTION_POINTERS result;
  std::memcpy(&result, exceptionInformation, sizeof(result));
  throw &result;
}
```

Результат работы программы:

```
Exception code: 0xc0000025
Exception flags: 0x0
Exception record: 0x00000000
Exception address: 0x77991812
Number parameters: 0

C:\SP\Debug\SP.exe процесс( 4572) завершает работу с кодом 2.
```

1.4.9 Использовать финальный обработчик finally

Помимо конструкции __try, __except поддерживается также конструкция __try, __finally. Блок __finally был создан для задачи высвобождения ресурсов и будет вызван **в любом случае** после завершения охраняемого кода [9] Докажем это, разработаем программу, которая выходит из охраняемого кода пятью различными способами:

```
#include <iostream>
#include <windows.h>

```

```
int main(const int argc, const char** argv) {
  if(argc != 2 || std::strlen(argv[1]) != 1)
    return 0x1;

  __try {
    std::cout << "Try block." << std::endl;

    switch(argv[1][0]) {
      case '0':
        // Самостоятельное завершение __try.
        break;

      case '1':
        // Нормальное заершение __try.
        __leave;

      case '2':
        // Безусловное завершение __try.
        goto out;

      case '3':
        // Завершение __try с исключением.
        RaiseException(EXCEPTION_DATATYPE_MISALIGNMENT, NULL, NULL, nullptr);
        break;

      case '4':
        // Завершение __try выходом из функции.
        return 0x0;

      default:
        return 0x2;
    }
  } __finally {
    std::cout << "Finally block." << std::endl;
  }

out:
  return 0x0;
}
```

Вне зависимости от способа выхода из охраняемого кода, блок __finally должен выполняться. Результат работы программы для всех пяти способов:

```
C
-- аргументом 0
Try block.
Finally block.C

-- аргументом 1
Try block.
Finally block.C

-- аргументом 2
Try block.
Finally block.C

-- аргументом 3
Try block.
Finally block.C

-- аргументом 4
Try block.
Finally block.
```

Таким образом блок __finally был вызван во всех пяти различных вариантах. В данном примере не были рассмотрены варианты выхода через break и continue, но результаты аналогичны.

1.4.10 Проверить корректность выхода из блока __try с помощью функции AbnormalTermination в финальном обработчике

Корректность выхода из охраняемого кода может повлиять на освобождение ресурсов в блоке __finally. Для этого была создана функция AbnormalTermination, имеющая следующую сигнатуру [10]:

```
BOOL AbnormalTermination(void);
```

Функция возвращает 0, если завершение нормальное и 1, если нет. Данная функция может быть вызвана только из блока __finally.

Корректным выходом из охраняемого кода считается самостоятельное завершение и команда __leave. Все остальные варианты выхода являются не нормальным завершением и должны вызывать раскрутку стека. Однако, MSVC от Microsoft добавляет оптимизацию выхода из охраняемого кода, не раскручивая стек (скорее всего используя __leave), что ускоряет работу кода, но паразитно для проведения экспериментов. Тем не мение, AbnormalTermination работает правильно, согласно спецификации.

Дополним программу из предыдущего пункта, добавив обработку нормального и не нормального завершения в блок __finally:

```
#include <iostream>
#include <windows.h>

int main(const int argc, const char** argv) {
  if(argc != 2 || std::strlen(argv[1]) != 1)
    return 0x1;

  __try {
    std::cout << "Try block." << std::endl;

    switch(argv[1][0]) {
      case '0':
        // Самостоятельное завершение __try.
        break;

      case '1':
        // Нормальное завершение __try.
        __leave;

      case '2':
        // Безусловное завершение __try.
        goto out;

      case '3':
        // Завершение __try с исключением.
        RaiseException(EXCEPTION_DATATYPE_MISALIGNMENT, NULL, NULL, nullptr);
        break;

      case '4':
        // Завершение __try выходом из функции.
        return 0x0;

      default:
        return 0x2;
    }
  } __finally {
    std::cout << "Finally block." << std::endl;

    if(AbnormalTermination()) {
      std::cout << "Abnormal termination." << std::endl;
    } else {
      std::cout << "Normal termination." << std::endl;
    }
  }

out:
  return 0x0;
```

```
}
```

Результат работы программы:

```
C
-- аргументом 0
Try block.
Finally block.
Normal termination.C

-- аргументом 1
Try block.
Finally block.
Normal termination.C

-- аргументом 2
Try block.
Finally block.
Abnormal termination.C

-- аргументом 3
Try block.
Finally block.
Abnormal termination.C

-- аргументом 4
Try block.
Finally block.
Abnormal termination.
```

Только в первых двух случаях (самостоятельное завершение и команда __leave) охраняемый код завершился нормально, в то время как в остальных случаях завершение было не нормальным.

1.5 Вывод

В результате работы были изучены структурные исключения SEH. Из преимуществ данного способа обработки исключений, по сравнению со встроенными средствами языка C++ является:

- возможность обработки аппаратных исключений и просмотра регистров процессора на момент их возникновения;
- поддерживаются как в языке C, так и в C++;
- возможность транслирования исключений в исключения языка C++.

Из минусов стоит отметить:

- зависимость от конкретной платформы, в то время как исключения языка C++ стандартизованы.

Кроме того, не стоит забывать, что раскрутка стека является достаточно трудоемкой операцией, поэтому программист должен отдавать себе отчет в том, какие операции вызывают нормальное завершение охраняемого кода, а какие нет. Стоит отметить, что иногда компилятор может встроить дополнительную оптимизацию, для того чтобы не раскручивать стек в таких ситуациях, однако, такие улучшения обычно не стандартизированы, не надежны и тяжело отслеживаются на этапе отладки.

1.6 Список литературы

[1] Эксплуатирование SEH в среде Win32 [Электронный ресурс]. — URL: http://www.securitylab.ru/contest/212085.php (дата обращения 05.10.2017).

[2] ОБРАБОТКА ИСКЛЮЧЕНИЙ В VISUAL C++ [Электронный ресурс]. — URL: http://www.avprog.narod.ru/progs/exceptions.htm (дата обращения 05.10.2017).

[3] Раскрутка стека. C++ для начинающих [Электронный ресурс]. — URL: https://it.wikireading.ru/35947 (дата обращения 05.10.2017).

[4] EXCEPTION_RECORD structure (Windows) [Электронный ресурс]. — URL: https://msdn.microsoft.com/ru-ru/library/windows/desktop/aa363082(v=vs.85).aspx (дата обращения 05.10.2017).

[5] RaiseException function (Windows) [Электронный ресурс]. — URL: https://msdn.microsoft.com/ru-ru/library/windows/desktop/ms680552(v=vs.85).aspx (дата обращения 05.10.2017).

[6] SetUnhandledExceptionFilter function (Windows) [Электронный ресурс]. — URL: https://msdn.microsoft.com/ru-ru/library/windows/desktop/ms680634(v=vs.85).aspx (дата обращения 05.10.2017).

[7] Оператор try-except [Электронный ресурс]. — URL: https://msdn.microsoft.com/ru-ru/library/s58ftw19.aspx (дата обращения 05.10.2017).

[8] Очень серьезный блог: Обработка исключений и корректность программ на C++. [Электронный ресурс]. — URL: http://evgeny-lazin.blogspot.ru/2008/07/blog-post.html (дата обращения 05.10.2017).

[9] Оператор try-finally [Электронный ресурс]. — URL: https://msdn.microsoft.com/ru-ru/library/9xtt5hxz.aspx (дата обращения 05.10.2017).

[10] AbnormalTermination macro (Windows) [Электронный ресурс]. — URL: https://msdn.microsoft.com/ru-ru/library/windows/desktop/ms679265(v=vs.85).aspx (дата обращения 05.10.2017).