Санкт-Петербургский государственный политехнический университет

Институт информационных технологий и управления

Кафедра компьютерных систем и программных технологий

**Отчёт лабораторной работе №3**

Обработка исключений

по дисциплине

Системное программное обеспечение

Работу выполнил студент группы № 53501/13 Дорофеев Юрий Владимирович

Работу принял преподаватель \_\_\_\_\_\_\_\_ Душутина Елена Владимировна

Санкт-Петербург

2013

**СОДЕРЖАНИЕ**

[1. ЗАДАНИЕ 3](#_Toc374032716)

[2. ВЫПОЛНЕНИЕ РАБОТЫ 4](#_Toc374032717)

[2.1. Сгенерировать и обработать исключение с помощь WinAPI функций 4](#_Toc374032718)

[2.1.1. Исключение EXCEPTION\_INT\_DIVIDE\_BY\_ZERO 4](#_Toc374032719)

[2.1.2. Исключение EXCEPTION\_FLT\_OVERFLOW 5](#_Toc374032720)

[2.2. Получить код исключения с помощью GetExceptionCode двумя способами 6](#_Toc374032721)

[2.2.1. Исключение EXCEPTION\_INT\_DIVIDE\_BY\_ZERO 6](#_Toc374032722)

[2.2.2. Исключение EXCEPTION\_FLT\_OVERFLOW 7](#_Toc374032723)

[2.3. Создать собственную функцию-фильтр 8](#_Toc374032724)

[2.3.1. Исключение EXCEPTION\_INT\_DIVIDE\_BY\_ZERO 8](#_Toc374032725)

[2.3.1. Исключение EXCEPTION\_FLT\_OVERFLOW 9](#_Toc374032726)

[2.4. Получить информацию об исключении с помощью GetExceptionInformation.Генерировать исключение с помощью RaiseException. 10](#_Toc374032727)

[2.4.1. Исключение EXCEPTION\_INT\_DIVIDE\_BY\_ZERO 12](#_Toc374032728)

[2.4.1. Исключение EXCEPTION\_FLT\_OVERFLOW 13](#_Toc374032729)

[2.5. Использовать функции UnhandleExceptionFilter и SetUnhandleExceptionFilter для необработанных исключений 14](#_Toc374032730)

[2.6. Обработать вложенные исключения 16](#_Toc374032731)

[2.7. Выйти из блока \_try с помощью goto и leave 18](#_Toc374032732)

[2.8. Преобразовать структурные исключения (\_try) в исключения языка С. Для этого использовать функцию-транслятор (translator) 20](#_Toc374032733)

[2.9. Использовать финальный обработчик finally 22](#_Toc374032734)

[2.10. Проверить корректность выхода из блока try с помощью функции AbnormalTermination в финальном обработчике finally 23](#_Toc374032735)

[ВЫВОДЫ 25](#_Toc374032736)

[СПИСОК ИСПОЛЬЗУЕМЫХ ИСТОЧНИКОВ 26](#_Toc374032737)

# ЗАДАНИЕ

1. Сгенерировать и обработать исключение с помощь WinAPI функций

1. Получить код исключения с помощью GetExceptionCode двумя способами:
   1. использовать эту функцию в выражении фильтре
   2. использовать эту функцию в обработчике
2. Создать собственную функцию-фильтр
3. Получить информацию об исключении с помощью GetExceptionInformation. генерировать исключение с помощью RaiseException.
4. Использовать функции InhandleExceptionFilter и SetUnhandleExceptionFilter для необработанных исключений
5. Обработать вложенные исключения
6. Выйти из блока \_try с помощью
7. goto
8. leave
9. Преобразовать структурные исключения (\_try) в исключения языка С. Для этого использовать функцию-транслятор (translator)
10. Использовать финальный обработчик finally
11. Проверить корректность выхода из блока try с помощью функции AbnormalTermination в финальном обработчике finally

# ВЫПОЛНЕНИЕ РАБОТЫ

## Сгенерировать и обработать исключение с помощь WinAPI функций

Исключением называется событие, которое произошло во время выполнения программы, в результате совершения которого дальнейшее нормальное выполнение программы становится невозможным. Как правило, такие события являются ошибками в программе. Поэтому для дальнейшей работы приложения требуется или восстановление программы в рабочее состояние, или ее аварийное завершение с освобождением всех захваченных программой ресурсов.

В операционных системах Windows для этой цели предназначен механизм структурной обработки исключений (structured exception handling, SEH). Смысл механизма структурной обработки исключений заключается в следующем. В программе выделяется блок программного кода, в котором может произойти исключение. Такой блок кода называется фреймом, а сам код называется охраняемым кодом. Затем, после фрейма вставляется программный блок, который обрабатывает происшедшее исключение. Этот блок называется обработчиком исключения. После обработки исключения управление передается первой инструкции, следующей за обработчиком исключения. Очевидно, что для того чтобы использовать этот механизм в программе, в язык программирования C++ нужно ввести новые ключевые слова. Такими ключевыми словами являются try и except, которые расширяют список стандартных ключевых слов языка программирования C++ и различаются только компилятором фирмы Microsoft. Ключевое слово try отмечает фрейм, а ключевое слово except отмечает обработчик исключения. В листинга 2.1 и 2.2 показаны простейшие примеры работы с исключениями.

### Исключение EXCEPTION\_INT\_DIVIDE\_BY\_ZERO

*Листинг 2.1* *Исключение EXCEPTION\_INT\_DIVIDE\_BY\_ZERO*

#include <windows.h>

#include <iostream>

using namespace std;

int main()

{

\_try

{

RaiseException( // вызываем исключение

3221225620, // код исключения деления на ноль

0, // continuable exception

0, NULL); // no arguments

}

\_except(EXCEPTION\_EXECUTE\_HANDLER)

{

DWORD ee = GetExceptionCode(); // получаем код исключения

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

cout << "EXCEPTION\_INT\_DIVIDE\_BY\_ZERO code is "<< ee <<endl;

else

cout << "Some other exception" << ee << endl;

}

return 0;

}

В результате выполнения, получили следующий результат:

EXCEPTION\_INT\_DIVIDE\_BY\_ZERO code is 3221225620

### Исключение EXCEPTION\_FLT\_OVERFLOW

Листинг 2.2 Исключение EXCEPTION\_FLT\_OVERFLOW

#include <windows.h>

#include <iostream>

using namespace std;

int main()

{

\_try

{

RaiseException(

3221225617, // код исключения

0, // continuable exception

0, NULL); // no arguments

}

\_except(EXCEPTION\_EXECUTE\_HANDLER)

{

DWORD ee = GetExceptionCode(); // получаем код исключения

if (ee == EXCEPTION\_FLT\_OVERFLOW)

cout << "EXCEPTION\_FLT\_OVERFLOW code is "<< ee <<endl;

else

cout << "Some other exception" << ee << endl;

}

return 0;

}

В результате выполнения, получили следующий результат:

EXCEPTION\_FLT\_OVERFLOW code is 3221225617

## Получить код исключения с помощью GetExceptionCode двумя способами

* 1. использовать эту функцию в выражении фильтре;
  2. использовать эту функцию в обработчике.

Пункт «б» был реализован выше.

Функция GetExceptionCode может вызываться только в выражении-фильтре или в блоке обработки исключения. Следовательно, эта функция вызывается всегда только в том случае, если исключение произошло. Отсюда можно определить назначение функции GetExceptionCode. Если эта функция вызывается в выражении фильтра, то она используется для того, чтобы определить выполняет ли текущий обработчик исключения обработку исключений с данным кодом или нужно продолжить поиск подходящего обработчика исключения. Если же функция GetExceptionCode вызывается в блоке обработки исключения, то она также предназначена для проверки кодов исключений, которые обрабатывает текущий обработчик исключения, но в этом случае поиск другого обработчика исключений не выполняется.

### Исключение EXCEPTION\_INT\_DIVIDE\_BY\_ZERO

Листинг 2.3 Исключение EXCEPTION\_INT\_DIVIDE\_BY\_ZERO

#include <windows.h>

#include <iostream>

using namespace std;

int main()

{

int a = 10;

int b = 0;

\_try

{

cout << "result = " << a/b << endl; // ошибка

}

\_except(GetExceptionCode() == EXCEPTION\_INT\_DIVIDE\_BY\_ZERO ? EXCEPTION\_EXECUTE\_HANDLER : EXCEPTION\_CONTINUE\_SEARCH)

{

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

}

return 0;

}

В результате выполнения, получили следующий результат:

EXCEPTION\_INT\_DIVIDE\_BY\_ZERO

### Исключение EXCEPTION\_FLT\_OVERFLOW

По умолчанию система отключает все исключения с плавающей точкой. Поэтому если при выполнении операции с плавающей точкой было получено число, которое не входит в диапазон представления чисел с плавающей точкой, то в результате система вернет NAN или INFINITY в случае слишком малого или слишком большого числа соответственно. Чтобы включить режим генерации исключений с плавающей точкой нужно изменить состояние слова, управляющего обработкой операций с плавающей точкой. Это можно сделать при помощи функции**\_controlfp**, которая имеет следующий прототип:

unsigned int \_controlfp(unsigned int new, unsigned int mask);

Прототип определен в заголовочном файле **float.h**. Эта функция возвращает старое слово, управляющее обработкой исключений. Параметр new задает новое управляющее слово, а параметр mask должен принимать значение \_MCW\_EM. Если значение этого параметра равно 0, то функция возвращает старое управляющее слово.

В параметре **new** для управления исключениями можно сбрасывать или устанавливать следующие значения:

* \_EM\_INVALID – исключение EXCEPTION\_FLT\_INVALID\_OPERATION;
* \_EM\_DENORMAL – исключение EXCEPTION\_FLT\_DENORMAL\_OPERAND;
* \_EM\_ZERODIVIDE – исключение EXCEPTION\_FLT\_DIVIDE\_BY\_ZERO;
* \_EM\_OVERFLOW – исключение EXCEPTION\_FLT\_OVERFLOW;
* \_EM\_UNDERFLOW – исключение EXCEPTION\_FLT\_UNDERFLOW;
* \_EM\_INEXACT – исключение EXCEPTION\_FLT\_INEXACT\_RESULT.

Если бит, соответствующий одному из этих исключений, сброшен, то система генерирует соответствующее исключение с плавающей точкой, иначе исключение не генерируется.

Листинг 2.4 Исключение EXCEPTION\_FLT\_OVERFLOW

#define \_CRT\_SECURE\_NO\_WARNINGS

#include <windows.h>

#include <iostream>

#include <float.h>

using namespace std;

int main()

{

int cw = \_controlfp(0, 0); //получить управляющее слово, заданное по умолчанию

//разрешить обработку исключений с плавающей точкой

cw &= ~(EM\_OVERFLOW | EM\_UNDERFLOW | EM\_INEXACT | EM\_ZERODIVIDE | EM\_DENORMAL);

\_controlfp(cw, \_MCW\_EM); //установить новое управляющее слово

\_try

{

double a = 0.01 / pow(99999, 999999);

}

\_except(GetExceptionCode() == EXCEPTION\_FLT\_OVERFLOW ? EXCEPTION\_EXECUTE\_HANDLER : EXCEPTION\_CONTINUE\_SEARCH)

{

cout << "EXCEPTION\_FLT\_OVERFLOW" <<endl;

}

return 0;

}

В результате выполнения, получили следующий результат:

EXCEPTION\_FLT\_OVERFLOW

## Создать собственную функцию-фильтр

Если для принятия решения об обработке исключения требуется более детально обработать информацию об исключении, то в выражении-фильтре используют функцию, которая в этом случае называется функцией фильтра. В функции фильтра Не разрешается вызывать функции GetExceptionCode и GetExceptionlnformation, однако эти функции могут вызываться для инициализации параметров функции фильтра.

Ниже приведены программы, в которых используется функция фильтра для принятия решения о дальнейшей обработке исключения. Функция фильтра возвращает одно из двух значений EXCEPTION\_CONTINUE\_EXECUTION ИЛИ EXCEPT ION\_EXECUTE\_HANDLER. Первое значение возвращается в случае исключения, которое генерируется системой при целочисленном делении на ноль, а второе — в остальных случаях.

### Исключение EXCEPTION\_INT\_DIVIDE\_BY\_ZERO

Листинг 2.5 Исключение EXCEPTION\_INT\_DIVIDE\_BY\_ZERO

include <windows.h>

#include <iostream>

using namespace std;

DWORD filter\_function(DWORD ee, int &b)

{

// проверяем код исключения

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

{

cout << "Integer divide by zero exception." << endl;

cout << "a = " << b << endl;

//восстанавливаем ошибку

b = 10;

// возобновляем выполнение программы

cout << "Continue execution." << endl;

//cout « "b = " « b « endl;

return EXCEPTION\_CONTINUE\_EXECUTION;

}

else

// прекращаем выполнение программы

return EXCEPTION\_EXECUTE\_HANDLER;

}

int main()

{

int a = 10;

int b = 0;

\_try

{

a = a/b; // ошибка

}

\_except(filter\_function(GetExceptionCode(),b))

{

cout << "Some exception" << endl;

}

return 0;

}

В результате выполнения, получили следующий результат:

Integer divide by zero exception.

b = 10

Continue execution.

### Исключение EXCEPTION\_FLT\_OVERFLOW

Листинг 2.6 Исключение EXCEPTION\_FLT\_OVERFLOW

#define \_CRT\_SECURE\_NO\_WARNINGS

#include <windows.h>

#include <iostream>

#include <float.h>

using namespace std;

DWORD filter\_function(DWORD ee, double &a)

{

// проверяем код исключения

if (ee == EXCEPTION\_FLT\_OVERFLOW)

{

cout << "EXCEPTION\_FLT\_OVERFLOW exception." << endl;

cout << "a = " << a << endl;

//восстанавливаем ошибку

a = 0;

// возобновляем выполнение программы

cout << "Continue execution." << endl;

return EXCEPTION\_CONTINUE\_EXECUTION;

}

else

// прекращаем выполнение программы

return EXCEPTION\_EXECUTE\_HANDLER;

}

int main()

{

double a;

int cw = \_controlfp(0, 0); //получить управляющее слово, заданное по умолчанию

//разрешить обработку исключений с плавающей точкой

cw &= ~(EM\_OVERFLOW | EM\_UNDERFLOW | EM\_INEXACT | EM\_ZERODIVIDE | EM\_DENORMAL);

\_controlfp(cw, \_MCW\_EM); //установить новое управляющее слово

\_try

{

a = 0.01 / pow(99999, 999999);

}

\_except(filter\_function(GetExceptionCode(),a))

{

cout << "Some exception" << endl;

}

return 0;

}

В результате выполнения, получили следующий результат:

EXCEPTION\_FLT\_OVERFLOW exception.

a = -9.25596e+061

Continue execution.

## Получить информацию об исключении с помощью GetExceptionInformation.Генерировать исключение с помощью RaiseException.

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

LPEXCEPTION\_POINTERS GetExceptionlnformation(VOID);

Эта функция возвращает указатель на структуру типа:

typedef struct EXCEPTION\_POINTERS {

PEXEPTION\_RECORD Except ionRecord;

PCONTEXT Context;

} EXCEPTION\_POINTERS, \* PEXCEPTION\_POINTERS;

которая, в свою очередь, содержит два указателя: ExceptionRecord и context на структуры типа exception\_record и context соответственно.

В структуру типа context система записывает содержимое всех регистров процессора на момент исключения. Эта структура имеет довольно громоздкое описание, которое можно найти в заголовочном файле WinNt.h.

Структура типа exception\_record имеет следующий формат:

typedef struct EXCEPTION\_RECORD {

DWORD Except i onCode;

DWORD ExceptionFlags,

strict \_EXCEPTION\_RECORD \*ExceptionRecord;

PVOID ExceptionAddress,

DWORD Number Parameters,

ULONG\_PTR Exceptionlnf ormation [ EXCEPTION\_MAXIMUM\_PARAMETERS ] ;

} EXCEPTION\_RECORD, \*PEXCEPTION\_RECORD;

В нее система записывает информацию об исключении. Поля этой структуры имеют следующее назначение. Поле ExceptionCode содержит код исключения, который может принимать такие же значения, как и код исключения, возвращаемый функцией GetExceptionCode.

Поле ExceptionFlags может принимать одно из двух значений:

* 0 — которое обозначает, что после обработки исключения возможно возобновление выполнения программы;
* EXCEPTION\_NONCONTINUABLE — которое обозначает, что после обработки исключения возобновление выполнения программы невозможно.

Если установлено значение EXCEPTION\_NONCONTINUABLE и выполнена попытка возобновления выполнения программы, то система выбросит исключение EXCEPTION\_NONCONTINUABLE\_EXCEPTION.

Поле ExceptionRecord содержит указатель на следующую структуру типа exception\_record, которая может быть создана в случае вложенных исключений.

Поле ExceptionAddress содержит адрес инструкции в программе, на которой произошло исключение.

Поле NumberParameters содержит количество параметров, заданных в поле Exceptioninformation, которое является последним в этой структуре.

Поле Exceptionlnformation[EXCEPTION\_MAXIMUM\_PARAMETERS] Определяет массив 32-битных аргументов, которые описывают исключение. Элементы этого массива могут использоваться функцией генерации программных исключений RaiseException. Отметим, что в операционных системах Windows NT/2000 для исключения с кодом exception\_access\_violation определены значения первых двух элементов этого массива. В этом случае первый элемент массива содержит одно из значений:

* 0 — попытка чтения виртуальной памяти;
* 1 — попытка записи в виртуальную память.

А второй элемент массива содержит адрес виртуальной памяти, по которому программа пыталась прочитать или записать данные.

Сделаем важное замечание о том, что функция GetExceptionlnformation может вызываться только в выражении фильтра. Поэтому эта функция вызывается всегда только в том случае, если исключение произошло. Кроме того, структуры типа exception\_pointers, exception\_record и context действительны только на время вычисления выражения-фильтра. Чтобы использовать содержимое структур типа exception\_record и context в блоке обработки исключения, его нужно сохранить в объявленных в программе переменных такого же типа. Как видно из описания структуры EXCEPTION\_RECORD, функцию GetExceptionlnformation можно использовать для двух целей: первая цель заключается в получении более подробной информации об исключении, учитывая содержимое структуры типа context; вторая цель состоит в обработке вложенных исключений.

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

### Исключение EXCEPTION\_INT\_DIVIDE\_BY\_ZERO

Листинг 2.7 Исключение EXCEPTION\_INT\_DIVIDE\_BY\_ZERO

#include <windows.h>

#include <iostream>

using namespace std;

EXCEPTION\_RECORD er; // информация об исключении

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

{

// сохраняем содержимое структуры EXCEPTION\_RECORD

er = \*(p->ExceptionRecord);

return EXCEPTION\_EXECUTE\_HANDLER;

}

int main()

{

\_try

{

RaiseException(

3221225620, // код исключения деления на ноль

0, // continuable exception

0, NULL); // no arguments

}

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

{

// распечатываем информацию об исключении

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

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

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

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

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

// распечатываем параметры

if (er.ExceptionCode == EXCEPTION\_INT\_DIVIDE\_BY\_ZERO)

{

cout << "Type of access = " << er.ExceptionInformation[0] << endl;

cout << "Address of access = " << er.ExceptionInformation[1] << endl;

}

cout << endl;

}

return 0;

}

В результате выполнения, получили следующий результат:

ExceptionCode = 3221225620

ExceptionFlags = 0

ExceptionRecord = 00000000

ExceptionAddress = 00F15F48

NumberParameters = 0

Type of access = 0

Address of access = 0

### Исключение EXCEPTION\_FLT\_OVERFLOW

Листинг 2.8 Исключение EXCEPTION\_FLT\_OVERFLOW

#include <windows.h>

#include <iostream>

using namespace std;

EXCEPTION\_RECORD er; // информация об исключении

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

{

// сохраняем содержимое структуры EXCEPTION\_RECORD

er = \*(p->ExceptionRecord);

return EXCEPTION\_EXECUTE\_HANDLER;

}

int main()

{

\_try

{

RaiseException( // вызываем исключение

3221225617, // код исключения деления на ноль

0, // continuable exception

0, NULL); // no arguments

}

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

{

// распечатываем информацию об исключении

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

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

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

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

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

// распечатываем параметры

if (er.ExceptionCode == EXCEPTION\_INT\_DIVIDE\_BY\_ZERO)

{

cout << "Type of access = " << er.ExceptionInformation[0] << endl;

cout << "Address of access = " << er.ExceptionInformation[1] << endl;

}

cout << endl;

}

return 0;

}

В результате выполнения, получили следующий результат

ExceptionCode = 3221225617

ExceptionFlags = 0

ExceptionRecord = 00000000

ExceptionAddress = 77AF2EEC

NumberParameters = 0

## Использовать функции UnhandleExceptionFilter и SetUnhandleExceptionFilter для необработанных исключений

Если в программе произошло исключение, для которого не существует обработчика исключений, то в этом случае вызывается функция-фильтр системного обработчика исключений, которая выводит на экран окно сообщений с предложением пользователю закончить программу аварийно или выполнить Отладку Приложения. Системная фуНКЦИЯ-филЬТр UnhandledExceptionFilter имеет следующий прототип:

LONG UnhandledExceptionFilter(PEXCEPTION\_POINTERS pExceptionlnfo);

Эта функция имеет один параметр, который указывает на структуру типа EXCEPTION\_INFO и возвращает одно из следующих значений:

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

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

LPTOP\_LEVEL\_EXCEPTION\_FILTER SetUnhandledExceptionFilter(

LPTOP\_LEVEL\_EXCEPTION\_FILTER lpTopLevelExceptionFilter

);

Эта функция возвращает адрес старой функции фильтра или null, если установлен системный обработчик исключений. Единственным параметром этой функции является указатель на новую функцию-фильтр, которая будет установлена вместо системной. Эта функция-фильтр должна иметь прототип, соответствующий системной функции фильтра UnhandledExceptionFilter, и возвращать одно из следующих значений:

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

Для того чтобы восстановить системную функцию-фильтр UnhandledExceptionFilter, НуЖНО Вызвать функцию UnhandledExceptionFilter с параметром null.

В листинге 2.8 приведена программа, которая устанавливает новую функцию-фильтр для необработанных исключений, а затем восстанавливает системную функцию UnhandledExceptionFilter.

Листинг 2.9 Работа с необработанными исключениями

#include <windows.h>

#include <iostream>

using namespace std;

LONG new\_filter(PEXCEPTION\_POINTERS pExceptionInfo)

{

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

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

return EXCEPTION\_EXECUTE\_HANDLER;

}

int main()

{

//RaiseException(

//3221225620, // код исключения деления на ноль

//0, // continuable exception

//0, NULL); // no arguments

LPTOP\_LEVEL\_EXCEPTION\_FILTER old\_filter;

// устанавливаем новую функцию-фильтр необработанных исключений

old\_filter = SetUnhandledExceptionFilter((LPTOP\_LEVEL\_EXCEPTION\_FILTER)new\_filter);

// выводим адрес старой функции-фильтра

cout << "Old filter-function address = " << hex << old\_filter << endl;

// создаем необработанное исключение

//\*p = 10;

return 0;

}

В результате выполнения, получили следующий результат

Old filter-function address = 0119130C

## Обработать вложенные исключения

При использовании структурной обработки исключений возможно вкладывать блоки try и except в другой блок try. В этом случае если функция-фильтр внутреннего блока except возвращает значение EXCEPTION\_CONTINUE\_SEARCH, то система удаляет все локальные объекты, принадлежащие текущим блокам try и except, и продолжает поиск обработчика исключений во внешних блоках try и except. Так как локальные объекты, определенные внутри любого блока, хранятся в стеке процесса, то фактически система очищает стек процесса. Область стека, которую занимают локальные объекты одного блока, называется фреймом стека. Поэтому можно сказать, что при обработке вложенных исключений выполняется очистка стека процесса от локальных объектов, определенных внутри вложенных блоков try и except. Такая очистка стека от локальных объектов называется глобальной раскруткой стека или просто раскруткой стека. В листинге 2.10 приведена программа, в которой используется обработка вложенных исключений.

Листинг 2.10 Обработка вложенных исключений

#include <windows.h>

#include <iostream>

using namespace std;

void f()

{

RaiseException(

3221225617, // код исключения

0, // continuable exception

0, NULL); // no arguments

}

void g()

{

RaiseException( // вызываем исключение

3221225620, // код исключения деления на ноль

0, // continuable exception

0, NULL); // no arguments

}

int main()

{

\_try

{

\_try

{

f();

}

\_except(GetExceptionCode() == EXCEPTION\_FLT\_OVERFLOW ?

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

{

cout << "Exception EXCEPTION\_FLT\_OVERFLOW." << endl;

}

\_try

{

g();

}

\_except(GetExceptionCode() == EXCEPTION\_FLT\_OVERFLOW ?

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

{

cout << "Exception EXCEPTION\_FLT\_OVERFLOW." << endl;

}

}

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

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

{

cout << "Exception integer divide by zero." << endl;

}

return 0;

}

В результате выполнения, получили следующий результат

Exception EXCEPTION\_FLT\_OVERFLOW.

Exception integer divide by zero.

Проведем анализ выполнения данной программы (рисунок 2.1.).

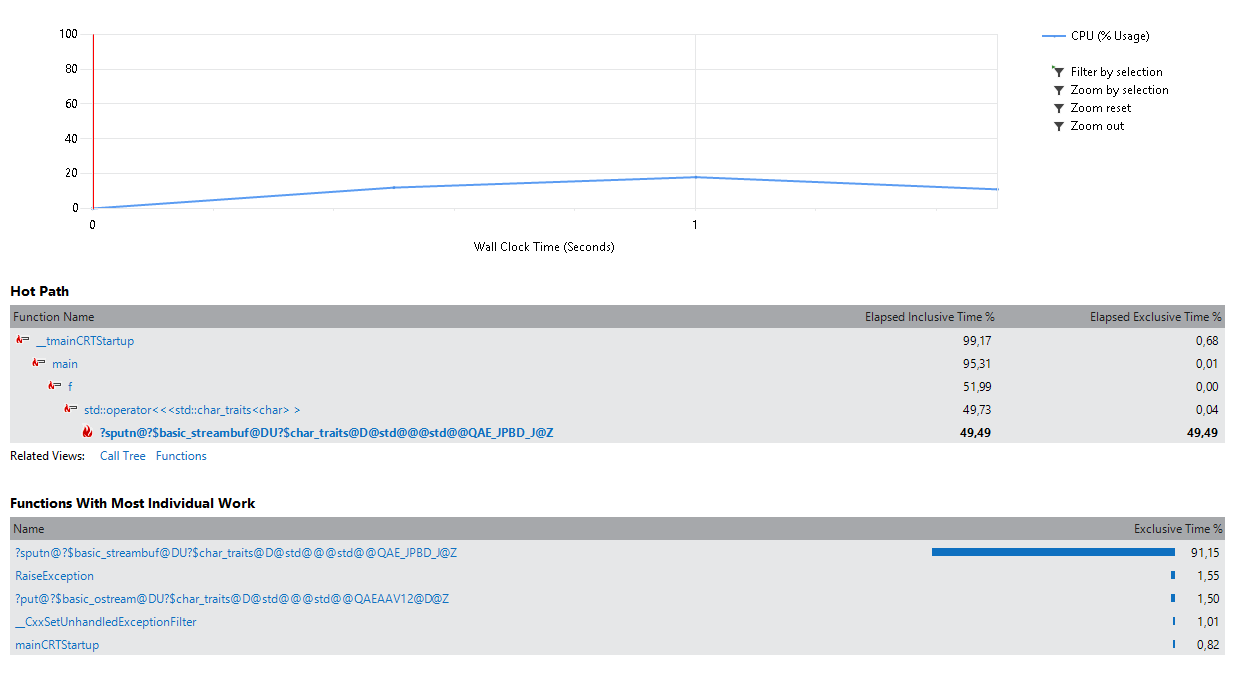


Рисунок . Анализ программы

По данным из отчета анализа производительности время выполнения данной программы = 2. 36 мс.

По данным рисунка 2.2 видно, что времени и ресурсы занимают 2 функции f – 52 % от общего времени выполнения, затем идет функция g 43%.

На рисунке 2.2 приведено полное дерево функций данной программы, на котором показано распределение ресурсов при ее выполнении.

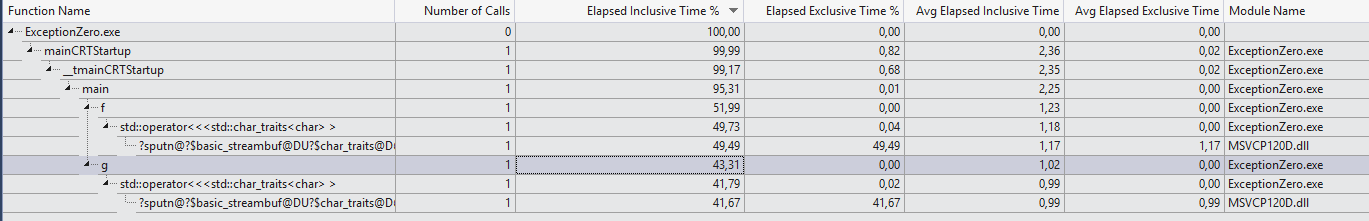


Рисунок . Древо функций

## Выйти из блока \_try с помощью goto и leave

Для передачи управления из фрейма можно использовать инструкцию goto языка программирования C++. В этом случае система считает, что блок с охраняемым кодом завершился аварийно и поэтому выполняет глобальную раскрутку стека. Следовательно, использование инструкции goto вызывает исполнение дополнительного программного кода, что замедляет выполнение программы. В листинге 2.11 приведена программа, которая используют инструкцию goto для выхода из блока try.

Листинг 2.11 Выход из блока \_try с помощью goto

#include <windows.h>

#include <iostream>

using namespace std;

int main()

{

int a = 10;

int b = 0;

\_try

{

if (b!=0)

{

cout << "Exit with goto from the try block." << endl;

goto exit\_try; // выходим их блока

}

else

b = 10;

}

\_except(EXCEPTION\_EXECUTE\_HANDLER)

{

cout << "There was some exception." << endl;

}

cout << "b = " << b << endl;

exit\_try: return 0;

}

В результате выполнения, получили следующий результат:

b = 10

Отметим, что в программе из листинга 2.11 выход из фрейма при помощи инструкции goto используется для обхода инструкции, следующей после блока except.

Если необходимо просто завершить выполнение блока try без аварийного выхода, т. е. не начиная раскрутку стека, то в этом случае нужно использовать инструкцию leave, которая введена в язык программирования C++ фирмой Microsoft. В листинге 2.12 приведена программа, в которой используется инструкция leave для выхода из блока try.

Листинг 2.12 Выход из блока \_try с помощью \_leave

#include <windows.h>

#include <iostream>

using namespace std;

int main()

{

int a = 10;

int b = 0;

\_try

{

if (b != 0)

{

cout << "Exit with leave from the try block." << endl;

\_leave; // выходим их блока

}

else

{

b = 10;

cout << "b = " << b << endl;

}

}

\_except(EXCEPTION\_EXECUTE\_HANDLER)

{

cout << "There was some exception." << endl;

}

return 0;

}

В результате выполнения, получили следующий результат:

b = 10

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

В реализации языка программирования C++ фирмой Microsoft, т. е. в Visual C++, предусмотрен механизм, который позволяет использовать механизм структурной обработки исключений в механизме обработки исключений, используемом в C++. Для этой цели была разработана функция \_set\_se\_transiator. Эта функция устанавливает в системе функцию, которая называется функцией-транслятором, назначение которой состоит в том, чтобы преобразовывать структурные исключения в исключения языка программирования C++. Если функция-транслятор установлена, то она вызывается всегда при выбросе структурного исключения. В функции-трансляторе можно использовать инструкцию throw языка программирования C++, которая будет выбрасывать исключение C++ нужного типа.

Функция-транслятор должна иметь следующий прототип:

typedef void (\*\_se\_translator\_function) (unsigned int, Struct \_EXCEPTION\_POINTERS\*);

который описан в заголовочном файле eh.h. Как видно из этого описания - функция-транслятор не возвращает значения и получает два параметра: код исключения и указатель на структуру типа \_EXCEPTION\_POINTERS.

Функция \_set\_se\_transiator, которая используется для установки функции-транслятора, также описана в заголовочном файле eh.h и имеет следующий прототип:

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

Единственным параметром этой функции является указатель на новую функцию-транслятор, а возвращает функция \_set\_se\_transiator адрес старой функции-транслятора, которая в дальнейшем может быть восстановлена при помощи вызова \_set\_se\_transiator. Если функция-транслятор устанавливается в первый раз, то возвращаемое значение может быть равно NULL.

В листинге 2.13 приведена программа, в которой устанавливается функция- транслятор. Эта функция просто выбрасывает исключение языка

Листинг 2.13 Выброс кода структурного исключения как исключения языка программирования C++

#include <windows.h>

#include <iostream>

#include <eh.h>

using namespace std;

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

{

throw code;

}

int main()

{

int a = 10, b = 0;

// устанавливаем функцию-транслятор

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

// перехватываем структурное исключение средствами C++

try

{

a /= b; // ошибка, деление на ноль

}

catch (unsigned code)

{

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

}

return 0;

}

Приведем в листинге 2.14 программу, которая выбрасывает структуру типа EXCEPTION\_POINTERS как исключение языка программирования C++.

Листинг 2.14 Выброс структуры типа EXCEPTION\_POINTERS как исключения языка программирования C++

#include <windows.h>

#include <iostream>

#include <eh.h>

using namespace std;

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

{

EXCEPTION\_RECORD er;

CONTEXT c;

EXCEPTION\_POINTERS ep = { &er, &c };

er = \*(info->ExceptionRecord);

c = \*(info->ContextRecord);

throw ep;

}

int main()

{

int a = 10, b = 0;

// устанавливаем функцию-транслятор

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

// перехватьшаем структурное исключение средствами C++

try

{

a /= b; // ошибка, деление на ноль

}

catch (EXCEPTION\_POINTERS ep)

{

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

}

return 0;

}

## Использовать финальный обработчик finally

В операционных системах Windows существует еще один способ обработки исключений, суть которого заключается в следующем. Код, при исполнении которого возможен выброс исключения, как и в случае с фреймовой обработкой исключений, заключается в блок try. Но только теперь за блоком try следует код, который заключается в блок finally. Система гарантирует, что при любой передаче управления из блока try, независимо от того, произошло или нет исключение внутри этого блока, предварительно управление будет передано блоку finally. Такой способ обработки исключений называется финальная обработка исключений. Структурно финальная обработка исключений выглядит следующим образом:

\_try

{

// охраняемый код

}

finally

{

// финальный код

}

Очевидно, что, как и в случае с фреймовой обработкой исключений, финальная обработка исключений требует поддержки как компилятора, так и операционной системы. Кроме того, в язык программирования C++ добавляется новое ключевое слово finally.

Финальная обработка исключений используется для того, чтобы при любом исходе исполнения блока try освободить ресурсы, которые были захвачены внутри этого блока. Такими ресурсами могут быть память, файлы, критические секции и т. д.

В листинге 2.15 приведена программа, в которой используется финальная обработка исключений.

Листинг 2.15 Финальная обработка исключений

#include <windows.h>

#include <iostream>

#include <eh.h>

using namespace std;

int main()

{

int a = 10;

int b = 0;

\_try

{

a = a / b;

}

\_finally

{

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

}

return 0;

}

В результате выполнения, получили следующий результат:

The finally block finished

Отметим, что блок finally выполняется при любом исходе выполнения блока try. Недостатком такой работы блока finally является то, что инструкция delete будет выполняться в любом случае, независимо от того, произошло исключение в блоке try или нет. Чтобы избежать такой ситуации, нужно проверить, как завершился блок try — нормально или нет.

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

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

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

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

BOOL AbnormaiTermination (VOID) ;

В случае если блок try завершился ненормально, эта функция возвращает ненулевое значение, а в противном случае — значение false. Используя функцию AbnormaiTermination, ресурсы, захваченные в блоке try, можно освобождать только в том случае, если блок try завершился ненормально. В листинге 2.16 приведена программа, которая использует функцию AbnormaiTermination для проверки нормального завершения блока try.

Листинг 2.16 Проверка нормального завершения блока \_try

#include <windows.h>

#include <iostream>

#include <eh.h>

using namespace std;

int main()

{

int a = 10;

int b = 0;

\_try

{

a = a / b;

}

\_finally

{

// если блок try закончился нормально

if (!AbnormalTermination())

{

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

}

else

// иначе нечего освобождать

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

}

return 0;

}

# ВЫВОДЫ

Механизм структурной обработки исключений в Windows немного отличается от механизма обработки исключений, принятого в языке программирования C++. Дело в том, что механизм структурной обработки исключений был разработан раньше, чем принят стандарт языка C++. Кроме того, в отличие от языка программирования C++ механизм структурной обработки исключений ориентирован не только на обработку программных исключений, но и на обработку аппаратных исключений. В SEH исключение рассматривается как ошибка, происшедшая при выполнении программы. В языке программирования C++ используется более абстрактный подход и исключение рассматривается как объект произвольного типа, который может выбросить программа, используя оператор throw. В свою очередь обработчик исключения catch может рассматриваться как функция с одним параметром, которая выполняется только в том случае, если тип ее параметра соответствует типу выброшенного исключения.

# СПИСОК ИСПОЛЬЗУЕМЫХ ИСТОЧНИКОВ

1. *ОБРАБОТКА ИСКЛЮЧЕНИЙ В VISUAL C++*. (б.д.). Получено из Программирование С++: http://www.avprog.narod.ru/progs/exceptions.htm
2. Побегайло, А. (2006). *Системное программирование в Windows.* Санкт-Петербург: БХВ-Петербург.