Обработка исключений в ОС Windows

Фреймовая и финальная обработки

Е.В.Душутина

Структурная обработка исключений Фреймовая обработка

Возможна: средствами языка C++,
средствами Windows (SEH) -
Structured Exception Handling (SEH)

Различия:

- SEH пригоден как для программных, так и для аппаратных исключений, в отличие от средств C++
- В SEH исключение – это ошибка при выполнении программы, в C++ - это объект произвольного типа, который может выбросить программа , используя оператор ***throw***. Обработчик исключения ***catch*** может рассматриваться как функция с одним параметром, кот. выполняется только при совпадении типа ее параметра с типом выброшенного исключения
- По-разному раскручивается стек

Суть механизма SEH

- В программе выделяется блок кода – фрейм, в котором может произойти исключение (охраняемый код)
- За фреймом размещается код обработчика
- Далее - первая инструкция, исполняемая после выполнения обработчика
- Используемые ключевые слова C++, различаемые компилятором MS: _try, _except для фрейма и обработчика соответственно

Структура программного фрагмента

```
_try
{  //охраняемый код
}
_except (выражение-фильтр)
{ //код обработки исключения
}
```

Значения (выражения-фильтра):

EXCEPTION_EXECUTE_HANDLER управление передается обработчику

EXCEPTION_CONTINUE_SEARCH продолжить поиск обработчика

EXCEPTION_EXECUTE_EXECUTION управление передается в точку прерывания программы

GetExceptionCode, GetExceptionInformation – функции для получения информации об исключении, м.б. использованны в выражении-фильтре

Не допускается в SEH: использование оператора ***goto*** для передачи управления внутрь фрейма или обработчика

Переменные, объявленные в фрейме и обработчике - **локальные**

Список значений, возвращаемых **GetExceptionCode**

Список констант находится в файле **WinBase.h**

- **EXCEPTION_ACCESS_VIOLATION** – поток попытался обратиться к виртуальному адресу, к которому у него нет доступа
- **EXCEPTION_BREAKPOINT** – выполнение достигло точки останова
- **EXCEPTION_DATATYPE_MISALIGNMENT** – попытка доступа к не выровненным данным на устройстве, которое требует выравнивания
- **EXCEPTION_SINGLE_STEP** – Возник сигнал о пошаговом выполнении программы (трассировочная ловушка или другой механизм)
- **EXCEPTION_ARRAY_BOUNDS_EXCEEDED** – обращение за границу массива на устройстве, которое проверяет границы
- **EXCEPTION_FLT_DENORMAL_OPERAND** – один из операндов операции с плавающей точкой не нормализован.
- **EXCEPTION_FLT_DIVIDE_BY_ZERO** – поток попытался сделать деление на ноль с плавающей точкой.
- **EXCEPTION_FLT_INEXACT_RESULT** - результат операции над числами с плавающей точкой нельзя точно представить в виде десятичной дроби
- **EXCEPTION_FLT_INVALID_OPERATION** – другие исключения операций с плавающей точкой, которые не выделены в данном списке
- **EXCEPTION_FLT_OVERFLOW** – переполнение при операции над числами с плавающей точкой.
- **EXCEPTION_FLT_STACK_CHECK** - переполнение стека или выход за его нижнюю границу в результате выполнения операции над числами с плавающей точкой
- **EXCEPTION_FLT_UNDERFLOW** - порядок результата операции над числами с плавающей точкой меньше минимальной величины для указанного типа данных
- **EXCEPTION_INT_DIVIDE_BY_ZERO** – целочисленное деление на ноль
- **EXCEPTION_INT_OVERFLOW** – переполнение разрядной сетки при операциях с целыми числами
- **EXCEPTION_PRIV_INSTRUCTION** – выполнение инструкции, которая не доступна в данном режиме процессора
- **EXCEPTION_NONCONTINUABLE_EXCEPTION** - фильтр исключений вернул**EXCEPTION_CONTINUE_EXECUTION** в ответ па невозобновляемое исключение (noncontinuable exception)

Применение функций GetExceptionCode и GetExceptionInformation

- Функции ***GetExceptionCode*** и ***GetExceptionInformation*** являются встраиваемыми, т.е. они поддерживаются компилятором напрямую и их нет ни в одной библиотеке.
- Их можно вызывать только из фильтра исключений, между скобками оператора _except, или из обработчика исключений, т.к. именно в этот момент существуют структуры с контекстом исключения,

```
__try
{
}
__except ((GetExceptionCode() ==
EXCEPTION_FLT_STACK_CHECK) ?    EXCEPTION_EXECUTE_HANDLER : EXCEPTION_CONTINUE_SEARCH)
{
    // обработка переполнения при операции с плавающей точкой
}
```

- Использование этих функций внутри фильтра недопустимо. Компилятор отслеживает все подобные попытки и выдает сообщения об ошибках. (см.след.слайд)

Функция-фильтр

- Если для принятия решения о правиле обработки исключения требуется более детальная работа с информацией, то используют не просто выражение-фильтр, а вызывают в этом выражении функцию-фильтр.
- Использование функций GetExceptionCode и GetExceptionInformation внутри фильтра недопустимо. Компилятор отслеживает все подобные попытки и выдает сообщения об ошибках. Однако эти функции могут вызываться при инициализации параметров функции фильтра.

Функция – фильтр

Пример вызова из обработчика

```
__try {
        }
 __except
{
       switch (GetExceptionCode())
            { …
             }
}
```

До этого использовали жестко заданные значения в качестве фильтра исключений.

Теперь заменим их на выражение с вызовом функции. Задача такой функции - проанализировать причину исключения, возможно, выполнить предварительную обработку, и вернуть значение, на основании которого будет выбираться обработчик исключения:

```
__try
{
          int a = 1 / 0;
}
__except (Filter(GetExceptionCode()))
{
}
LONG Filter(DWORD dwExceptionGode)
{
          return((dwExceptionCode == EXCEPTION_ACCESS_VIOLATION) ? EXCEPTION_EXECUTE_HANDLER :
EXCEPTION_CONTINUE_SEARCH);
}
```

Функции Filter выполняется **на этапе поиска обработчика исключения**

Если возникает необходимость использовать GetExceptionInformation или GetExceptionCode из функции фильтра исключений, надо передавать результат их работы через параметр фильтра (как в примере)

функция *GetExceptionInformation*

- При возникновении исключения, система запоминает структуры **EXCEPTION_RECORD, CONTEXT и EXCEPTION_POINTERS**. Они в совокупности описывают полностью контекст возникновения исключений.
- Для доступа к информации из структур **EXCEPTION_RECORD** или **CONTEXT** надо сохранить их содержимое из фильтра исключения. Эти структуры существуют только во время обработки фильтра исключения, поэтому запоминать на них указатель нет смысла.
- Из структуры **EXCEPTION_RECORD** можно получить код исключения, адрес возникновения исключения и другую полезную информацию. Это все позволяет гибко строить стратегии обработки ошибок.

В частности, в совокупности с мини дампами можно построить систему сбора информации об ошибках на стороне клиента, чтобы своевременно вносить исправления в код и добиваться высокого качества ПО.

Генерация программных исключений

- Программные исключения могут вызываться принудительно самой программой
- они могут использоваться для сообщения об ожидаемых ошибках, таких, как ошибки при доступе к файлу, некорректный ввод пользователя и т.п.
- для сообщения о фатальных ошибках, которые в конечном итоге приведут к завершению программы
- генерируются функцией ***RaiseException***, которая имеет следующий прототип:

```
void RaiseException( OWORD dwExceptionCode, DWORD dwExceptionFlags, DWORD
    nNumberOfArguments, CONST ULONG_PTR *pArguments);
```

ПАРАМЕТРЫ:

dwExceptionCode - это код исключения. Его можно получить с помощью GetExceptionCode. Код исключения можно построить по правилам, применяемым для стандартных кодов ошибок Windows. Формат кода ошибки определен в файле **winerror.h**

dwExceptionFlags указывает, можно ли возобновить выполнение программы с команды, следующей за ***RaiseException***. Он может принимать значения 0 или EXCEPTION_NONCONTINUABLE. Если вы указали EXCEPTION_NONCONTINUABLE, то фильтр обработчика исключения не может вернуть значение EXCEPTION_CONTINUE_EXECUTION. Если фильтр все же вернет это значение, то система возбудит еще одно исключение с кодом EXCEPTION_NONCONTINUABLE_EXCEPTION.

Параметры **NumberParameters** и **ExceptionInformation** могут использоваться для передачи параметров исключения. Они доступны обработчику через структуру EXCEPTION_RECORD. Обычно эти параметры нулевые. Если все же надо передать параметры, то nNumberOfArguments содержит количество указателей ULONG_PTR в массиве pArguments.

Пример вызова *RaiseException*

```
RaiseException(STATUS_NO_MEMORY,
EXCEPTION_NONCONTINUABLE,  0, NULL);
```

// Сообщаем о том, что в системе нет памяти, и запрещаем перезапуск приложения со следующей инструкции. Предполагаем, что обработчики исключения не в состоянии изыскать недостающую память, т.е. перезапуск не имеет смысла.

Необработанные исключения

- Если произошло исключение, непредусмотренное программой (обработчика не существует), то вызывается функция-фильтр системного обработчика с предложением аварийно закончить приложение или выполнить его отладку.
- Прототип системной функции:

```
LONG WINAPI UnhandledExceptionFilter(
  _In_  struct _EXCEPTION_POINTERS *ExceptionInfo
);
```

Параметр ExceptionInfo [in] указывает на структуру типа EXCEPTION_POINTERS, где содержится описание исключения и контекст процессора на момент исключения (то, что возвращает ф-ция GetExceptionInformation)

- Возвращаемые значения: передать управление
  EXCEPTION_CONTINUE_SEARCH - отладчику приложения
  EXCEPTION_EXECUTE_HANDLER – обработчику исключения

Замена системной функции-фильтра

С помощью функции:

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

Функция возвращает адрес старой функции-фильтра или NULL, если установлен системный обработчик исключений.

Параметр – указатель на новую функцию-фильтр, она должна иметь прототип, соответствующий системной функции-фильтру ***UnhandledExceptionFilter***

Новая функция-фильтр должна возвращать одно из значений:

EXCEPTION_EXECUTE_HANDLER – прекращение выполнения программы

EXCEPTION_CONTINUE_EXECUTION – возобновление выполнения с точки исключения

EXCEPTION_CONTINUE_SEARCH – выполняется системная функция ***UnhandledExceptionFilter***

Для восстановления системной функции нужно вызвать функцию ***UnhandledExceptionFilter*** с параметром ***NULL***

Обработка вложенных исключений

- Возможно вложение блоков **_try** и **_except** в другой блок **_try**. Если функция-фильтр внутреннего _except возвращает EXCEPTION_CONTINUE_SEARCH, то система удаляет все локальные объекты, принадлежащие текущим блокам _try и _except и хранящиеся в стеке процесса, и продолжает поиск обработчика исключений во внешних блоках _try и _except, т.е. фактически система очищает стек процесса.
- Фрейм стека – область стека, занимаемая локальными объектами одного блока.
- При обработке вложенных исключений выполняется **глобальная раскрутка стека** или просто **раскрутка стека**, т.е. очистка стека процесса от локальных объектов, определенных внутри вложенных блоков.

Передача управления и выход из фрейма

- Инструкция C++ ***goto :*** система считает, что блок завершился аварийно и выполняет **глобальную** раскрутку стека (исполнение дополнительного программного кода, замедление выполнения программы) – подход часто используется для обхода инструкции, следующей после блока _except (свидетельствует о плохой структурированности программы).
- Инструкция MS C++ ***_leave*** – без аварийного выхода, не начиная раскрутки стека.

Встраивание SEH в механизм исключений C++

- В Visual C++ есть функция _set_se_translator, преобразующая структурные исключения (SEH) в исключения C++.
- В функции-трансляторе можно использовать инструкцию ***throw*** (C++), кот. будет выбрасывать исключение C++ нужного типа.
- Прототип (C++, C#):

typedef void (*_se_translator_function)(unsigned int, struct _EXCEPTION_POINTERS*);

описан в заголовочном файле eh.h.

Ничего не возвращает, получает 2 параметра: код исключения и указатель на структуру _EXCEPTION_POINTERS.

- Функция для установки функции-транслятора

se_translator_function

_set_se_translator(_se_translator_function seTransFunction);

Параметр – указатель на новую функцию-транслятор,

Возвращает адрес старой функции, которую можно восстановить вызовом этой же функции **_set_se_translator** (если функция устанавливается впервые? возвращаемое значение может быть NULL)

Финальная обработка исключений

- Еще один способ обработки исключений
- Структура финальной обработки

```
__try
{
  //Охраняемый код
}
__finally
{
  //Финальный код
}
```

Финальная обработка исключений используется для того, чтобы при любом исходе исполнения блока __try освободить ресурсы (память, файлы, критические секции и т.п.), которые были захвачены внутри этого блока.

Финальный код будет выполняться **в любом** случае. Во избежание ошибок необходимо проверять завершение блока _try – нормальное или нет.

Проверка завершения фрейма

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

- Нормальное завершение блока;
- Выход из блока при помощи управляющей инструкции __leave – нормальное завершение;
- Выход из блока при помощи одной из управляющих инструкций return, break, continue или goto – **ненормальное** завершение блока;
- Передача управления обработчику исключения -– **ненормальное** завершение блока.

Чтобы определить, как завершился блок __try, используется функция:

BOOL **AbnormalTermination**(VOID);

возвращает ненулевое значение, если завершение ненормальное, иначе – FALSE.

Используя эту функцию, ресурсы, захваченные в блоке __try, можно освобождать только, если блок __try завершился ненормально.

Обработка вложенных финальных блоков

- Можно вкладывать блоки **_try** и **_finally** в другой блок **_try**.
- Если внутри самого внутреннего блока произошло исключение, то выполняется **раскрутка стека** (как и при фреймовой обработке)
- При раскрутке стека управление передается всем вложенным блокам **_finally** в порядке, обратном их вложенности.

Задание для практического выполнения

Выполнить задание для исключения: ошибка деления на ноль

Выполнить для заданного (выбранного) исключения

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