Системное программирование

Лекция 8

Структурная обработка исключений

План лекции

- Структурная обработка исключений
- Обработчики завершения
- Фильтры и обработчики исключений
- Необработанные исключения и исключения C++

Структурная обработка исключений

Закроем глаза и помечтаем, какие бы программы мы писали, если бы сбои в них были невозможны!

Представляете: памяти навалом, неверных указателей никто не передает, нужные файлы всегда на месте

Не программирование, а праздник, да? А код программ? Насколько он стал бы проще и понятнее! Без всех этих if и goto

И если Вы давно мечтали о такой среде программирования, Вы сразу же оцените **структурную обработку исключений**

Преимущество SEH в том, что при написании кода можно сосредоточиться на решении своей задачи. Если при выполнении программы возникнут неприятности, система сама обнаружит их и сообщит пользователю

Структурная обработка исключений

Но что за неприятности такие? Это – исключения!

Исключение – это событие, возникающее из-за выполнения определенной команды, которая вызвала ошибку процессора

Скорее всего в результате такого события нормальное выполнение программы становится **не возможным**!

Исключения в некотором роде похожи на прерывания, основное отличие заключается в том, что исключение является **синхронным** и **технически воспроизводимым** при тех же условиях, в то время как прерывание является асинхронным и может произойти в любой момент

Примеры исключений: деление на ноль, точка останова, ошибка страницы, переполнение стека и недопустимая инструкция

Структурная обработка исключений

Если возникает исключение, ядро перехватывает его и позволяет коду обработать исключение, если это возможно

Этот механизм и называется **Structured Exception Handling** (**SEH**) и доступен как для кода пользовательского режима, так и для кода режима ядра

Для справки! SEH является частью **исключительно** операционной системы Windows! Также стоит отметить, что полная поддержка SEH присутствует только в компиляторе **MSVC**!

Структурная обработка исключений

Хотя всю работу по отлову исключений берёт на себя операционная система, однако основная нагрузка по поддержке SEH ложится на компилятор, а не на операционную систему

Он генерирует специальный код на входах и выходах **блоков исключений** (**exception blocks**), создает таблицы вспомогательных структур данных для поддержки SEH и предоставляет функции обратного вызова, к которым система могла бы обращаться для прохода по блокам исключений

Компилятор отвечает и за формирование **стековых фреймов** (**stack frames**) и другой внутренней информации, используемой операционной системой. **Стековым фреймом** называется область стека, которую занимают локальные объекты одного блока

Данные понятия нам понадобятся далее при обсуждении понятия раскрутки стека!

Структурная обработка исключений

SEH предоставляет две основные возможности: **обработку завершения** (**termination handling**) и **обработку исключений** (**exception handling**)

Не путайте SEH с обработкой исключений в C++, которая представляет собой еще одну форму обработки исключений, построенную на применении ключевых слов языка C++ **catch** и **throw**. При этом Microsoft Visual C++ использует преимущества поддержки SEH, уже обеспеченной компилятором и операционными системами Windows

Несколько подробнее данный механизм будет рассмотрен позже в данной лекции

Структурная обработка исключений

Забегая наперёд стоит привести список ключевых слов используемых для работы с SEH в MSVC:

Ключевое слово	Описание
__try	Начинает блок кода, в котором могут возникать исключения
__except	Указывает, обработано ли исключение, и предоставляет код обработки, если это так
__finally	Предоставляет код, который гарантированно будет выполнен независимо от того, завершается ли блок __try обычным образом, с помощью инструкции return или из-за исключения
__leave	Предоставляет оптимизированный механизм для перехода к блоку __finally откуда-либо из блока __try

Структурная обработка исключений

Собственно, **обработчик завершения (__finally)** гарантирует, что блок кода (собственно обработчик) будет выполнен независимо от того, как происходит выход из другого блока кода – защищенного участка программы. Синтаксис обработчика завершения при работе с компилятором MSVC выглядит так:

```
__try {
    // защищенный блок
    ⋮
}
__finally {
    // обработчик завершения
    ⋮
}
```

Структурная обработка исключений

В предыдущем фрагменте кода совместные действия операционной системы и компилятора гарантируют, что код блока **__finally** обработчика завершения будет выполнен независимо от того, как произойдет выход из **защищенного блока**

И неважно, разместите Вы в защищенном блоке операторы **return** или **goto** – обработчик завершения все равно будет вызван!!!

Кстати, а что такое **защищенный блок?**

Защищенный или охраняемый блок кода – это блок кода, ограниченный фигурными скобками оператора **__try**. Предполагается, что в этом блоке может возникнуть исключение, которое следует обработать

Структурная обработка исключений

Пронумерованные комментарии подсказывают, в каком порядке будет выполняться этот код.
Использование в Funcenstein1 блоков **try-finally** на самом деле мало что дает
Код ждет освобождения семафора, изменяет содержимое защищенных данных, сохраняет новое значение в локальной переменной ***dwTemp***, освобождает семафор и возвращает новое значение тому, кто вызвал эту функцию

```
DWORD
Funcenstein1()
{
  DWORD dwTemp;
  // 1. Что-то делаем здесь
  //...
  __try
  {
    // 2. Запрашиваем разрешение на доступ
    // к защищенным данным, а затем используем их
    WaitForSingleObject(g_hSem, INFINITE);
    g_dwProtectedData = 5;
    dwTemp = g_dwProtectedData;
  }
  __finally
  {
    // 3. Даем и другим попользоваться защищенными данными
    ReleaseSemaphore(g_hSem, 1, NULL);
  }
  // 4. Продолжаем что-то делать
  return (dwTemp);
}
```

Структурная обработка исключений

В конец блока **__try** в функции Funcenstein2 добавлен оператор **return**. Он сообщает компилятору, что Вы хотите выйти из функции и вернуть значение переменной ***dwTemp*** (в данный момент равное 5) Но, если будет выполнен **return**, текущий поток никогда не освободит семафор, и другие потоки не получат шанса занять этот семафор. Такой порядок выполнения грозит вылиться в действительно серьезную проблему: ведь потоки, ожидающие семафора, могут оказаться не в состоянии возобновить свое выполнение

```
DWORD
Funcenstein2()
{
  DWORD dwTemp;
  // 1. Что-то делаем здесь
  //...
  __try
  {
    // 2. Запрашиваем разрешение на доступ
    // к защищенным данным, а затем используем их
    WaitForSingleObject(g_hSem, INFINITE);
    g_dwProtectedData = 5;
    dwTemp = g_dwProtectedData;
    // возвращаем новое значение
    return (dwTemp);
  }
  __finally
  {
    // 3. Даем и другим попользоваться защищенными данными
    ReleaseSemaphore(g_hSem, 1, NULL);
  }
  // продолжаем что-то делать - в данной версии
  // этот участок кода никогда не выполняется
  dwTemp = 9;
  return (dwTemp);
}
```

Структурная обработка исключений

Применив обработчик завершения, мы не допустили преждевременного выполнения оператора **return**. Когда **return** пытается реализовать выход из блока **__try**, компилятор проверяет, чтобы сначала был выполнен код в блоке **__finally**, – причем до того, как оператору **return** в блоке **__try** будет позволено реализовать выход из функции

Вызов ReleaseSemaphore в обработчике завершения (в функции Funcenstein2) гарантирует освобождение семафора – поток не сможет случайно сохранить права на семафор и тем самым лишить процессорного времени все ожидающие этот семафор потоки

Структурная обработка исключений

После выполнения блока **__finally** функция фактически завершает работу. Любой код за блоком **__finally** не выполняется, поскольку возврат из функции происходит внутри блока **__try**
Каким же образом компилятор гарантирует выполнение блока **__finally** до выхода из блока **__try**?
Дело вот в чем:
Просматривая исходный текст, компилятор видит, что Мы вставили **return** внутрь блока **__try.** Тогда он генерирует код, который сохраняет возвращаемое значение (в нашем примере 5) в созданной им же временной переменной. Затем создает код для выполнения инструкций, содержащихся внутри блока **__finally**, – это называется **локальной раскруткой** (**local unwind**)
По сути раскрутка это процесс освобождения локальных объектов каждого из блоков из стека процесса (в частности вложенных блоков)

Структурная обработка исключений

Точнее, **локальная раскрутка** происходит, когда система выполняет блок **__finally** из-за **преждевременного** выхода из блока **__try**. Значение временной переменной, сгенерированной компилятором, возвращается из функции после выполнения инструкций в блоке **__finally**

Как видите, чтобы все это вытянуть, компилятору приходится генерировать дополнительный код, а системе – выполнять дополнительную работу. На разных типах процессоров поддержка обработчиков завершения реализуется по-разному, вплоть до **сотен тысяч дополнительных** машинных команд, что может отрицательно сказаться на быстродействии программы

Поэтому лучше не писать код, вызывающий преждевременный выход из блока **__try** обработчика завершения

Структурная обработка исключений

Обработка исключений предназначена для перехвата тех исключений, которые происходят не слишком часто (в нашем случае – преждевременного возврата)

Если же какое-то исключение – чуть ли не норма, гораздо эффективнее проверять его явно, не полагаясь на SEH.

Заметьте: когда поток управления выходит из блока **__try** естественным образом (как в Funcenstein1), издержки от вызова блока **__finally** минимальны, так как для входа в **__finally** при **нормальном** выходе из **__try** исполняется ***всего одна*** машинная команда – вряд ли Вы заметите ее влияние на быстродействие своей программы

Структурная обработка исключений

Обнаружив в блоке **__try** функции Funcenstein3 оператор **goto**, компилятор генерирует код для локальной раскрутки, чтобы сначала выполнялся блок **__finally**. Но на этот
раз после **__finally** исполняется код, расположенный за меткой ReturnValue, так как возврат из функции не происходит ни в блоке **__try**, ни в блоке **__finally**. В итоге функция возвращает 5. И опять, поскольку Вы прервали естественный ход потока управления
из **__try** в **__finally**, быстродействие программы – в зависимости от типа процессора – может снизиться весьма значительно

```
DWORD
Funcenstein3()
{
  DWORD dwTemp;
  // 1. Что-то делаем здесь
  //...
  __try
  {
    // 2. Запрашиваем разрешение на доступ
    // к защищенным данным, а затем используем их
    WaitForSingleObject(g_hSem, INFINITE);
    g_dwProtectedData = 5;
    dwTemp = g_dwProtectedData;
    // пытаемся перескочить через блок finally
    goto ReturnValue;
  } __finally {
    // 3. Даем и другим попользоваться защищенными данными
    ReleaseSemaphore(g_hSem, 1, NULL);
  }
  dwTemp = 9;
// 4. Продолжаем что-то делать
ReturnValue:
  return (dwTemp);
}
```

Структурная обработка исключений

Допустим, в функции **Funcinator**, вызванной из блока **__try**, – «жучок», из-за которого возникает нарушение доступа к памяти. Без SEH пользователь в очередной раз увидел бы самое известное диалоговое окно Application Error
Стоит его закрыть – завершится и приложение. Если бы процесс завершился (из-за неправильного доступа к памяти), семафор остался бы занят – соответственно и ожидающие его потоки не получили бы процессорное время. Но вызов ReleaseSemaphore в блоке **__finally** гарантирует освобождение семафора, даже если нарушение доступа к памяти происходит в какой-то другой функции

```
DWORD
Funcfurter1()
{
  DWORD dwTemp;
  // 1. Что-то делаем здесь
  //...
  __try
  {
    // 2. Запрашиваем разрешение на доступ
    // к защищенным данным, а затем используем их
    WaitForSingleObject(g_hSem, INFINITE);
    dwTemp = Funcinator(g_dwProtectedData);
  }
  __finally
  {
    // 3. Даем и другим попользоваться защищенными данными
    ReleaseSemaphore(g_hSem, 1, NULL);
  }
  // 4. Продолжаем что-то делать
  return (dwTemp);
}
```

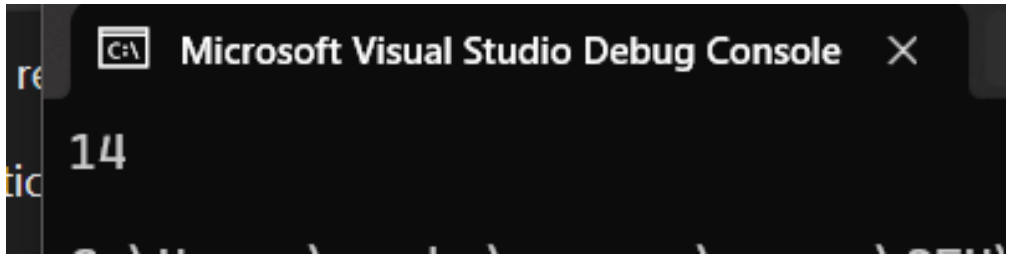
Структурная обработка исключений

Проверим себя!

Что вернёт следующая функция?

```
DWORD
FuncaDoodleDoo()
{
  DWORD dwTemp = 0;
  while (dwTemp < 10) {
    __try {
      if (dwTemp == 2)
        continue;
      if (dwTemp == 3)
        break;
    } __finally {
      dwTemp++;
    }
    dwTemp++;
  }
  dwTemp += 10;
  return (dwTemp);
}
```

Правильный ответ:



Почему?

Структурная обработка исключений

Блок **__try** в Funcenstein4 пытается вернуть значение переменной dwTemp (5) функции, вызвавшей Funcenstein4
Как мы уже отметили при обсуждении Funcenstein2, попытка преждевременного возврата из блока **__try** приводит к генерации кода, который записывает возвращаемое значение во временную переменную, созданную компилятором. Затем выполняется код в блоке **__finally**
Funcenstein4 является копией Funcenstein2, но с добавлением в блок **__finally** оператора return
Вопрос: что вернет Funcenstein4 – 5 или 103?

```
DWORD
Funcenstein4()
{
  DWORD dwTemp;
  // 1. Что-то делаем здесь
  //...
  __try
  {
    // 2. Запрашиваем разрешение на доступ
    // к защищенным данным, а затем используем их
    WaitForSingleObject(g_hSem, INFINITE);
    g_dwProtectedData = 5;
    dwTemp = g_dwProtectedData;
    // возвращаем новое значение
    return (dwTemp);
  }
  __finally
  {
    // 3. Даем и другим попользоваться защищенными данными
    ReleaseSemaphore(g_hSem, 1, NULL);
    return (103);
  }
  // продолжаем что-то делать – этот код
  // никогда не выполняется
  dwTemp = 9;
  return (dwTemp);
}
```

Структурная обработка исключений

Итак, **обработчики завершения**, хоть и весьма эффективные однако при преждевременном выходе из блока **__try**, могут дать **нежелательные** результаты именно потому, что предотвращают досрочный выход из блока **__try**

Лучше всего избегать любых операторов, способных вызвать преждевременный выход из блока **__try** обработчика завершения. А в идеале – удалить все операторы **return**, **continue**, **break**, **goto** (и им подобные) как из блоков **__try**, так и из блоков **__finally**

Тогда компилятор сгенерирует код и более компактный (перехватывать преждевременные выходы из блоков **__try** не понадобится), и более быстрый (на локальную раскрутку потребуется меньше машинных команд). Да и читать Ваш код будет гораздо легче

Структурная обработка исключений

Теперь поговорим о том, как обработчики завершения упрощают более сложные задачи программирования. Взгляните на функцию, в которой не используются преимущества обработки завершения

Проверки ошибок в функции Funcarama1 затрудняют чтение ее текста, что усложняет ее понимание, сопровождение и модификацию

```
BOOL
Funcarama1()
{
  HANDLE hFile = INVALID_HANDLE_VALUE;
  PVOID pvBuf = NULL;
  DWORD dwNumBytesRead;
  BOOL fOk;
  hFile = CreateFile("SOMEDATA.DAT",
                     GENERIC_READ,
                     FILE_SHARE_READ,
                     NULL,
                     OPEN_EXISTING,
                     0,
                     NULL);
  if (hFile == INVALID_HANDLE_VALUE) {
    return (FALSE);
  }
  pvBuf = VirtualAlloc(NULL, 1024, MEM_COMMIT, PAGE_READWRITE);
  if (pvBuf == NULL) {
    CloseHandle(hFile);
    return (FALSE);
  }
  fOk = ReadFile(hFile, pvBuf, 1024, &dwNumBytesRead, NULL);
  if (!fOk || (dwNumBytesRead == 0)) {
    VirtualFree(pvBuf, 1024, MEM_RELEASE | MEM_DECOMMIT);
    CloseHandle(hFile);
    return (FALSE);
  }
  // что-то делаем с данными
  //...
  // очистка всех ресурсов
  VirtualFree(pvBuf, 1024, MEM_RELEASE | MEM_DECOMMIT);
  CloseHandle(hFile);
  return (TRUE);
}
```

Структурная обработка исключений

Конечно, можно переписать Funcarama1 так, чтобы она стала яснее
Funcarama2 легче для понимания, но по-прежнему трудна для модификации и сопровождения
Кроме того, приходится делать слишком много отступов по мере добавления новых условных операторов; после такой переделки Вы того и гляди начнете писать код на правом краю экрана и переносить операторы на другую строку
через каждые пять символов!

```
BOOL
Funcarama2()
{
  HANDLE hFile = INVALID_HANDLE_VALUE;
  PVOID pvBuf = NULL;
  DWORD dwNumBytesRead;
  BOOL fOk, fSuccess = FALSE;
  hFile = CreateFile("SOMEDATA.DAT",
                     GENERIC_READ,
                     FILE_SHARE_READ,
                     NULL,
                     OPEN_EXISTING,
                     0,
                     NULL);
  if (hFile != INVALID_HANDLE_VALUE) {
    pvBuf = VirtualAlloc(NULL, 1024, MEM_COMMIT, PAGE_READWRITE);
    if (pvBuf != NULL) {
      fOk = ReadFile(hFile, pvBuf, 1024, &dwNumBytesRead, NULL);
      if (fOk && (dwNumBytesRead != 0)) {
        // что-то делаем с данными
        //...
        fSuccess = TRUE;
      }
    }
    VirtualFree(pvBuf, 1024, MEM_RELEASE | MEM_DECOMMIT);
  }
  CloseHandle(hFile);
  return (fSuccess);
}
```

Структурная обработка исключений

Перепишем-ка еще раз первый вариант (Funcarama1), задействовав преимущества обработки завершения
Главное достоинство Funcarama3 в том, что весь код, отвечающий за очистку, собран в одном месте – в блоке **__finally**. Если понадобится включить что-то в эту функцию, то для очистки мы просто добавим одну-единственную строку в блок **__finally** – возвращаться к каждому месту возможного возникновения ошибки и вставлять в него строку для очистки не нужно

```
BOOL
Funcarama3()
{
  // Внимание: инициализируйте все переменные, предполагая худшее
  HANDLE hFile = INVALID_HANDLE_VALUE;
  PVOID pvBuf = NULL;
  __try {
    DWORD dwNumBytesRead;
    BOOL fOk;
    hFile = CreateFile("SOMEDATA.DAT",
                       GENERIC_READ,
                       FILE_SHARE_READ,
                       NULL,
                       OPEN_EXISTING,
                       0,
                       NULL);
    if (hFile == INVALID_HANDLE_VALUE) {
      return (FALSE);
    }
    pvBuf = VirtualAlloc(NULL, 1024, MEM_COMMIT, PAGE_READWRITE);
    if (pvBuf == NULL) {
      return (FALSE);
    }
    fOk = ReadFile(hFile, pvBuf, 1024, &dwNumBytesRead, NULL);
    if (!fOk || (dwNumBytesRead != 1024)) {
      return (FALSE);
    }
    // что-то делаем с данными
    //...
  } __finally {
    // очистка всех ресурсов
    if (pvBuf != NULL)
      VirtualFree(pvBuf, 1024, MEM_RELEASE);
    if (hFile != INVALID_HANDLE_VALUE)
      CloseHandle(hFile);
  }
  // продолжаем что-то делать
  return (TRUE);
}
```

Структурная обработка исключений

Настоящая проблема в Funcarama3 – расплата за изящество. Уже говорилось: избегайте по возможности операторов **return** внутри блока **__try**
Чтобы облегчить последнюю задачу, Microsoft ввела еще одно ключевое слово в свой компилятор C: **__leave.** Вот новая версия (Funcarama4), построенная на применении нового ключевого слова

```
BOOL
Funcarama4()
{
  // Внимание: инициализируйте все переменные, предполагая худшее
  HANDLE hFile = INVALID_HANDLE_VALUE;
  PVOID pvBuf = NULL;
  // предполагаем, что выполнение функции будет неудачным
  BOOL fFunctionOk = FALSE;
  __try {
    DWORD dwNumBytesRead;
    BOOL fOk;
    hFile = CreateFile("SOMEDATA.DAT",
                       GENERIC_READ,
                       FILE_SHARE_READ,
                       NULL,
                       OPEN_EXISTING,
                       0,
                       NULL);
    if (hFile == INVALID_HANDLE_VALUE) {
      __leave;
    }
    pvBuf = VirtualAlloc(NULL, 1024, MEM_COMMIT, PAGE_READWRITE);
    if (pvBuf == NULL) {
      __leave;
    }
    fOk = ReadFile(hFile, pvBuf, 1024, &dwNumBytesRead, NULL);
    if (!fOk || (dwNumBytesRead == 0)) {
      __leave;
    }
    // что-то делаем с данными
    //...
    // функция выполнена успешно
    fFunctionOk = TRUE;
  } __finally {
    // очистка всех ресурсов
    if (pvBuf != NULL)
      VirtualFree(pvBuf, MEM_RELEASE | MEM_DECOMMIT);
    if (hFile != INVALID_HANDLE_VALUE)
      CloseHandle(hFile);
  }
  // продолжаем что-то делать
  return (fFunctionOk);
}
```

Структурная обработка исключений

Ключевое слово **__leave** в блоке **__try** вызывает переход в конец этого блока. Можно рассматривать это как переход на закрывающую фигурную скобку блока **__try**.

И никаких неприятностей это не влечёт, потому что выход из блока **__try** и вход в блок **__finally** происходит **естественным образом**

Правда, нужно ввести дополнительную булеву переменную fFunctionOk, сообщающую о завершении функции: удачно оно или нет. Но это дает минимальные издержки

Структурная обработка исключений

Разрабатывая функции, использующие обработчики завершения делайте именно так, инициализируйте все описатели ресурсов недопустимыми значениями перед входом в блок **__try**. Тогда в блоке **__finally** Вы проверите, какие ресурсы выделены успешно, и узнаете тем самым, какие из них следует потом освободить

Другой распространенный метод отслеживания ресурсов, подлежащих освобождению, – установка флага при успешном выделении ресурса. Код **__finally** проверяет состояние флага и таким образом определяет, надо ли освобождать ресурс

Структурная обработка исключений

Итак, пока нам с Вами удалось четко выделить только два сценария, которые приводят к выполнению блока **__finally:**

- нормальная передача управления от блока **__try** блоку **__finally**
- **локальная раскрутка** – преждевременный выход из блока **__try** (из-за операторов goto, continue, break, return и т. д.), вызывающий принудительную передачу управления блоку **__finally**

Третий сценарий – **глобальная раскрутка** (**global unwind**) – протекает не столь выраженно. Вспомним Funcfurter1. Ее блок **__try** содержал вызов функции Funcinator. При неверном доступе к памяти в Funcinator глобальная раскрутка приводила к выполнению блока **__finally** в Funcfurter1

Структурная обработка исключений

Выполнение кода в блоке **__finally** всегда начинается в результате возникновения одной из этих трех ситуаций. Чтобы определить, какая из них вызвала выполнение блока **__finally**, вызовите встраиваемую функцию AbnormalTermination:

```
BOOL AbnormalTermination();
```

Её можно вызвать только из блока **__finally**; она возвращает булево значение, которое сообщает, был ли преждевременный выход из блока **__try**, связанного с данным блоком **__finally**. Иначе говоря, если управление естественным образом передано из **__try** в **__finally**, **AbnormalTermination** возвращает FALSE. А если выход был преждевременным – то вызов **AbnormalTermination** дает TRUE

Структурная обработка исключений

Следующий фрагмент демонстрирует использование встраиваемой функции **AbnormalTermination**

```
DWORD
Funcfurter2()
{
  DWORD dwTemp;
  // 1. Что-то делаем здесь
  //...
  __try {
    // 2. Запрашиваем разрешение на доступ
    // к защищенным данным, а затем используем их
    WaitForSingleObject(g_hSem, INFINITE);
    dwTemp = Funcinator(g_dwProtectedData);
  } __finally {
    // 3. Даем и другим попользоваться защищенными данными
    ReleaseSemaphore(g_hSem, 1, NULL);
    if (!AbnormalTermination()) {
      // в блоке try не было ошибок - управление
      // передано в блок finally естественным образом
      //...
    } else {
      // что-то вызвало исключение, и, так как в блоке try
      // нет кода, который мог бы вызвать преждевременный
      // выход, блок finally выполняется из-за глобальной
      // раскрутки
      // если бы в блоке try был оператор goto, мы бы
      // не узнали, как попали сюда
      //...
    }
  }
  // 4. Продолжаем что-то делать
  return (dwTemp);
}
```

Структурная обработка исключений

Теперь Вы знаете, как создавать обработчики завершения. Давайте суммируем причины, по которым следует применять обработчики завершения.

- Упрощается обработка ошибок – очистка гарантируется и проводится в одном месте
- Улучшается восприятие текста программ
- Облегчается сопровождение кода
- Удается добиться минимальных издержек по скорости и размеру кода — при условии правильного применения обработчиков

Структурная обработка исключений

Исключение – это событие, которого Вы не ожидали

В хорошо написанной программе не предполагается попыток обращения по неверному адресу или деления на нуль. И все же такие ошибки случаются. За перехват попыток обращения по неверному адресу и деления на нуль отвечает центральный процессор, возбуждающий исключения в ответ на эти ошибки

Исключение, возбужденное процессором, называется **аппаратным** (**hardware exception**)

Также операционная система и прикладные программы способны возбуждать собственные исключения – **программные** (**software exceptions**)

Структурная обработка исключений

При возникновении аппаратного или программного исключения операционная система дает Вашему приложению шанс определить его тип и самостоятельно обработать
Синтаксис **обработчика исключений** таков:

```
__try {
    // защищенный блок
    ⋮
}
__except (фильтр исключений) {
    // обработчик исключений
    ⋮
}
```

Структурная обработка исключений

Обратите внимание на ключевое слово **__except**. За блоком **__try** всегда должен следовать либо блок **__finally**, либо блок **__except**

Для данного блока **__try** нельзя указать одновременно и блок **__finally**, и блок **__except**; к тому же за **__try** не может следовать несколько блоков **__finally** или **__except**

Однако **try-finally** можно вложить в **try-except**, и наоборот

В отличие от обработчиков завершения, **фильтры** и **обработчики исключений** выполняются непосредственно операционной системой – нагрузка на компилятор при этом минимальна

Структурная обработка исключений

В блоке **__try** функции Funcmeister1 мы просто присваиваем 0 переменной ***dwTemp***.
Такая операция не приведет к исключению, и поэтому код в блоке **__except** никогда не выполняется
Обратите внимание на такую особенность: конструкция **try-finally** ведет себя иначе. После того как переменной ***dwTemp*** присваивается 0, следующим исполняемым оператором оказывается **return**

```
DWORD
Funcmeister1()
{
  DWORD dwTemp;
  // 1. Что-то делаем здесь
  //...
  __try
  {
    // 2. Выполняем какую-то операцию
    dwTemp = 0;
  }
  __except (EXCEPTION_EXECUTE_HANDLER)
  {
    // обрабатываем исключение;
    // этот код никогда не выполняется
    //...
  }
  // 3. Продолжаем что-то делать
  return (dwTemp);
}
```

Структурная обработка исключений

Хотя ставить операторы **return**, **goto**, **continue** и **break** в блоке **__try** обработчика завершения настоятельно не рекомендуется, их применение в этом блоке не приводит к снижению быстродействия кода или к увеличению его размера

Использование этих операторов в блоке **__try**, связанном с блоком **__except**, не вызовет таких неприятностей, как локальная раскрутка

Структурная обработка исключений

Инструкция внутри блока **__try** функции **Funcmeister2** пытается поделить 5 на 0. Перехватив это событие, процессор возбуждает аппаратное исключение. Тогда операционная система ищет начало блока **__except** и проверяет выражение, указанное в качестве фильтра исключений; оно должно дать один из **трех идентификаторов**, определенных в заголовочном Windows-файле Excpt.h

```
DWORD
Funcmeister2()
{
  DWORD dwTemp = 0;
  // 1. Что-то делаем здесь
  //...
  __try
  {
    // 2. Выполняем какую-то операцию
    dwTemp = 5 / dwTemp; // генерирует исключение
    dwTemp += 10;        // никогда не выполняется
  }
  __except (/* 3. Проверяем фильтр */ EXCEPTION_EXECUTE_HANDLER)
  {
    // 4. Обрабатываем исключение
    MessageBeep(0);
    //...
  }
  // 5. Продолжаем что-то делать
  return (dwTemp);
}
```

Структурная обработка исключений

Фильтры исключений:
- EXCEPTION_EXECUTE_HANDLER
- EXCEPTION_CONTINUE_SEARCH
- EXCEPTION_CONTINUE_EXECUTION

EXCEPTION_EXECUTE_HANDLER – это значение сообщает системе в основном вот что: «Я вижу это исключение; так и знал, что оно где-нибудь произойдет; у меня есть код для его обработки, и я хочу его сейчас выполнить.»

В этот момент система проводит глобальную раскрутку, а затем управление передается коду внутри блока **__except** (коду обработчика исключений). После его выполнения система считает исключение обработанным и разрешает программе продолжить работу

Структурная обработка исключений

Но вот откуда возобновится выполнение?

Приложение возобновляет выполнение с инструкции, следующей за блоком **__except**. По окончании выполнения кода в блоке **__except** управление передается на первую строку за этим блоком

Когда фильтр исключений возвращает **EXCEPTION_EXECUTE_HANDLER**, системе приходится проводить **глобальную раскрутку**. Она приводит к продолжению обработки всех незавершенных блоков **try-finally**, выполнение которых началось вслед за блоком **try-except**, обрабатывающим данное исключение

Структурная обработка исключений

```
void
FuncOStimpy1()
{
  // 1. Что-то делаем здесь
  //...
  __try {
    // 2. Вызываем другую функцию
    FuncORen1();
    // этот код никогда не выполняется
  } __except (/* 6. Проверяем фильтр исключений */
      EXCEPTION_EXECUTE_HANDLER) {
    // 8. После раскрутки выполняется этот обработчик
    MessageBox(...);
  }
  // 9. Исключение обработано - продолжаем выполнение
  //...
}
```

```
void
FuncORen1()
{
  DWORD dwTemp = 0;
  // 3. Что-то делаем здесь
  //...
  __try
  {
    // 4. Запрашиваем разрешение на доступ к защищенным данным
    WaitForSingleObject(g_hSem, INFINITE);
    // 5. Изменяем данные, и здесь генерируется исключение
    g_dwProtectedData = 5 / dwTemp;
  }
  __finally
  {
    // 7. Происходит глобальная раскрутка, так как
    // фильтр возвращает EXCEPTION_EXECUTE_HANDLER
    // Даем и другим попользоваться защищенными данными
    ReleaseSemaphore(g_hSem, 1, NULL);
  }
  // сюда мы никогда не попадем
  //...
}
```

Структурная обработка исключений

Глобальную раскрутку, осуществляемую системой, можно остановить, если в блок **__finally** включить оператор **return**
Заметьте: код блока **__except** в **FuncMonkey** никогда не вызовет **MessageBeep**.
Оператор **return** в блоке **__finally** функции **FuncPheasant** заставит систему вообще прекратить раскрутку, и поэтому выполнение продолжится так, будто ничего не произошло

```
void
FuncMonkey()
{
  __try {
    FuncFish();
  } __except (EXCEPTION_EXECUTE_HANDLER) {
    MessageBeep(0);
  }
  MessageBox(...);
}
void
FuncFish()
{
  FuncPheasant();
  MessageBox(...);
}
void
FuncPheasant()
{
  __try {
    strcpy(NULL, NULL);
  } __finally {
    return;
  }
}
```

Структурная обработка исключений

EXCEPTION_CONTINUE_EXECUTION – обнаружив такое значение выражения в фильтре, система возвращается к инструкции, вызвавшей исключение, и пытается выполнить ее снова

EXCEPTION_CONTINUE_SEARCH – данный идентификатор указывает системе перейти к предыдущему блоку **__try**, которому соответствует блок **__except**, и обработать его фильтр

Это значит, что система пропускает при просмотре цепочки блоков любые блоки **__try**, которым соответствуют блоки **__finally** (а не **__except**). Причина этого очевидна: в блоках **__finally** нет фильтров исключений, а потому и проверять в них нечего

Структурная обработка исключений

Часто фильтр исключений должен проанализировать ситуацию, прежде чем определить, какое значение ему вернуть. Например, Ваш обработчик может знать, что делать при делении на нуль, но не знать, как обработать нарушение доступа к памяти. Именно поэтому фильтр отвечает за анализ ситуации и возврат соответствующего значения

```
__try {
    x = 0;
    y = 4 / x;
} __except ((GetExceptionCode() == EXCEPTION_INT_DIVIDE_BY_ZERO)
                ? EXCEPTION_EXECUTE_HANDLER
                : EXCEPTION_CONTINUE_SEARCH) {
    // обработка деления на нуль
}
```

Структурная обработка исключений

Встраиваемая функция **GetExceptionCode** возвращает идентификатор типа исключения (некоторые из кодов описаны **здесь**)

Встраиваемую функцию **GetExceptionCode** можно вызвать только из фильтра исключений (между скобками, которые следуют за **__except**) или из обработчика исключений

Однако **GetExceptionCode** нельзя вызывать из функции фильтра исключений. Компилятор помогает вылавливать такие ошибки и обязательно сообщит о таковой

Структурная обработка исключений

Коды исключений формируются по тем же правилам, что и коды ошибок, определенные в файле WinError.h. Каждое значение типа DWORD разбивается на поля

Биты	31–30	29	28	27–16	15–0
Содержимое:	Код степени «тяжести» (severity)	Кем определен — Microsoft или пользователем	Зарезерви-рован	Код подсистемы (facility code)	Код исключения
Значение:	0 = успех 1 = информация 2 = предупреждение 3 = ошибка	0 = Microsoft 1 = пользователь	Должен быть 0 (см. таблицу ниже)	Определяется Microsoft	Определяется Microsoft или пользовате-лем

По сути данная структура исключения соответствует структуре HRESULT из методологии COM

Структурная обработка исключений

Когда возникает исключение, операционная система заталкивает в стек соответствующего потока структуры EXCEPTION_RECORD, CONTEXT и EXCEPTION_POINTERS

EXCEPTION_RECORD содержит информацию об исключении, независимую от типа процессора, а CONTEXT – машинно-зависимую информацию об этом исключении.

В структуре EXCEPTION_POINTERS всего два элемента – указатели на помещенные в стек структуры EXCEPTION_RECORD и CONTEXT

```
typedef struct _EXCEPTION_POINTERS {
  PEXCEPTION_RECORD ExceptionRecord;
  PCONTEXT          ContextRecord;
} EXCEPTION_POINTERS, *PEXCEPTION_POINTERS;
```

Структурная обработка исключений

Чтобы получить эту информацию и использовать ее в программе, вызовите **GetExceptionInformation**

Эта встраиваемая функция возвращает указатель на структуру EXCEPTION_POINTERS

Самое важное в **GetExceptionInformation** то, что ее можно вызывать только в фильтре исключений и больше нигде, потому что структуры CONTEXT, EXCEPTION_RECORD и EXCEPTION_POINTERS существуют лишь во время обработки фильтра исключений. Когда управление переходит к обработчику исключений, эти данные в стеке разрушаются

Структурная обработка исключений

До сих пор мы рассматривали обработку аппаратных исключений, когда процессор перехватывает некое событие и возбуждает исключение

Но Вы можете и сами генерировать исключения. Это еще один способ для функции сообщить о неудаче вызвавшему ее коду

Традиционно функции, которые могут закончиться неудачно, возвращают некое особое значение – **признак ошибки**

При этом предполагается, что код, вызвавший функцию, проверяет, не вернула ли она это особое значение, и, если да, выполняет какие-то альтернативные операции

Структурная обработка исключений

Альтернативный подход заключается в том, что при неудачном вызове функции **возбуждают исключения**

Тогда написание и сопровождение кода становится гораздо проще, а программы работают намного быстрее

Последнее связано с тем, что та часть кода, которая отвечает за контроль ошибок, вступает в действие лишь при сбоях, т. е. в исключительных ситуациях

Возбудить программное исключение несложно – достаточно вызвать функцию **RaiseException**

Структурная обработка исключений

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

Ее первый параметр, ***dwExceptionCode***, – значение, которое идентифицирует генерируемое исключение

Второй параметр функции – ***dwExceptionFlags*** – должен быть либо 0, либо EXCEPTION_NONCONTINUABLE. В принципе этот флаг указывает, может ли фильтр исключений вернуть EXCEPTION_CONTINUE_EXECUTION в ответ на данное исключение

Третий и четвертый параметры (***nNumberOfArguments*** и ***pArguments***) функции **RaiseException** позволяют передать дополнительные данные о генерируемом исключении

Структурная обработка исключений

Собственные программные исключения генерируют в приложениях по целому ряду причин

Например, чтобы посылать информационные сообщения в системный журнал событий. Как только какая-нибудь функция в Вашей программе столкнется с той или иной проблемой, Вы можете вызвать **RaiseException**; при этом обработчик исключений следует разместить выше по дереву вызовов, тогда – в зависимости от типа исключения – он будет либо заносить его в журнал событий, либо сообщать о нем пользователю

Вполне допустимо возбуждать программные исключения и для уведомления о внутренних фатальных ошибках в приложении

Структурная обработка исключений

Мы обсудили, что происходит, когда фильтр возвращает значение EXCEPTION_CONTINUE_SEARCH. Оно заставляет систему искать дополнительные фильтры исключений, продвигаясь вверх по дереву вызовов. А что будет, если все фильтры вернут EXCEPTION_CONTINUE_SEARCH? Тогда мы получим **необработанное исключение** (**unhandled exception**)

Для таких случаев может быть вызвана особая функция фильтра, предоставляемая операционной системой:

```
LONG UnhandledExceptionFilter(PEXCEPTION_POINTERS pExceptionInfo);
```

Структурная обработка исключений

Она выводит окно, указывающее на то, что поток в процессе вызвал необрабатываемое им исключение, и предлагает либо закрыть процесс, либо начать его отладку

```
VOID
BaseThreadStart(PTHREAD_START_ROUTINE pfnStartAddr, PVOID pvParam)
{
  __try {
    ExitThread((pfnStartAddr)(pvParam));
  } __except (UnhandledExceptionFilter(GetExceptionInformation())) {
    ExitProcess(GetExceptionCode());
  }
  // Примечание: сюда мы никогда не попадем
}
```

Структурная обработка исключений

Для изменения стандартного поведения функции **UnhandledExceptionFilter** можно вызвать функцию:

```
PTOP_LEVEL_EXCEPTION_FILTER SetUnhandledExceptionFilter(
   PTOP_LEVEL_EXCEPTION_FILTER pTopLevelExceptionFilter);
```

После ее вызова необработанное исключение, возникшее в любом из потоков процесса, приведет к вызову Вашего фильтра исключений. Адрес фильтра следует передать в единственном параметре функции **SetUnhandledExceptionFilter**. Прототип этой функции-фильтра должен выглядеть так:

```
LONG UnhandledExceptionFilter(PEXCEPTION_POINTERS pExceptionInfo);
```

Структурная обработка исключений

Что лучше использовать: SEH или исключения C++?

Для начала позвольте напомнить, что SEH – механизм операционной системы, доступный в любом языке программирования, а исключения C++ поддерживаются только в C++

Создавая приложение на C++, Вы должны использовать средства именно этого языка, а не SEH. Причина в том, что исключения C++ – часть самого языка и его компилятор автоматически создает код, который вызывает деструкторы объектов и тем самым обеспечивает корректную очистку ресурсов

Структурная обработка исключений

Что лучше использовать: SEH или исключения C++?

Однако Вы должны иметь в виду, что компилятор **MSVC** реализует обработку исключений C++ на основе SEH операционной системы

Например, когда Вы создаете C++-блок **try**, компилятор генерирует SEH-блок **__try**

C++-блок **catch** становится SEH-фильтром исключений, а код блока **catch** – кодом SEH-блока **__except**

По сути, обрабатывая C++-оператор **throw**, компилятор генерирует вызов Windows-функции **RaiseException**, и значение переменной, указанной в **throw**, передается этой функции как дополнительный аргумент

Структурная обработка исключений

```
void ChunkyFunky() {
   try {
      // тело блока try
      ⋮
      throw 5;

   }
   catch (int x) {


      // тело блока catch
      ⋮
   }
   ⋮
}
```

```
void ChunkyFunky() {
   __try {
      // тело блока try
      ⋮
      RaiseException(Code=0xE06D7363,
         Flag=EXCEPTION_NONCONTINUABLE,Args=5);
   }
   __except ((ArgType == Integer) ?
      EXCEPTION_EXECUTE_HANDLER :
      EXCEPTION_CONTINUE_SEARCH) {
      // тело блока catch
      ⋮
   }
   ⋮
}
```

Структурная обработка исключений

Что лучше использовать: SEH или исключения C++?

Следует отметить, что иногда для обработки исключений механизм SEH встраивают в стандартный механизм **try/throw/catch** языка C++

Это возможно благодаря функции **_set_se_translator**. Она позволяет установить функцию преобразования SEH-исключений в C++-исключения

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

Структурная обработка исключений

Практические рекомендации

- Используйте SEH для критических операций, таких как работа с файлами или сетью
- Всегда документируйте возможные исключения и способы их обработки
- Избегайте подавления исключений без необходимости
- Тестируйте сценарии, связанные с возникновением ошибок

Системное программирование

Лекция 8

Структурная обработка исключений