**Вопросы для контроля:**

1. **Что такое исключение?**

**Исключение** – это событие, возникающее из-за выполнения определенной команды, которая вызвала ошибку процессора. В результате такого события нормальное выполнение программы становится невозможным. Исключение является синхронным и технически воспроизводимым при тех же условиях. Примеры включают деление на ноль, ошибку доступа к памяти и другие. Исключения бывают аппаратными (возбуждаются процессором) и программными (возбуждаются ОС или приложениями).

1. **Что такое SEH?**

**SEH** (Structured Exception Handling - Структурная обработка исключений) – это механизм операционной системы Windows, который позволяет перехватывать исключения и дает коду возможность их обработать, если это возможно. Он доступен как для кода пользовательского режима, так и для кода режима ядра. Полная поддержка SEH присутствует только в компиляторе MSVC.

1. **Что такое обработка завершений?**

**Обработка завершений** (termination handling) – это одна из двух основных возможностей SEH, предоставляемая ключевым словом \_\_finally. Обработчик завершения (\_\_finally) гарантирует, что блок кода (собственно обработчик) будет выполнен независимо от того, как происходит выход из защищенного блока (\_\_try) – будь то нормальное завершение, преждевременный выход (return, goto и т.п.) или возникновение исключения.

1. **Что такое обработка исключений?**

**Обработка исключений** (exception handling) – это вторая основная возможность SEH, предоставляемая ключевым словом \_\_except с фильтром. Она предназначена для перехвата и реагирования на произошедшие исключения. При возникновении исключения система ищет соответствующий блок \_\_except, проверяет его фильтр и в зависимости от значения фильтра выполняет код обработчика исключений или продолжает поиск.

1. **Что такое защищенный блок кода?**

**Защищенный или охраняемый блок кода** – это блок кода, ограниченный фигурными скобками оператора \_\_try. Предполагается, что в этом блоке может возникнуть исключение, которое следует обработать (с помощью \_\_except) или для которого требуется гарантированное выполнение кода очистки (\_\_finally).

1. **Для чего нужно ключевое слово \_\_leave?**

Ключевое слово \_\_leave используется в блоке \_\_try (в связке с \_\_finally). Оно предоставляет оптимизированный механизм для перехода к блоку \_\_finally из любой точки внутри блока \_\_try. В отличие от операторов return, goto, break, continue, которые вызывают дорогую локальную раскрутку при выходе из \_\_try в \_\_finally, \_\_leave вызывает переход в конец \_\_try естественным образом, минимизируя издержки и позволяя гарантированно выполнить \_\_finally.

1. **Что такое локальная и глобальная раскрутки?**

**Локальная раскрутка** (local unwind) – это процесс, который происходит при преждевременном выходе из блока \_\_try (например, из-за return, goto, break, continue) в блок \_\_finally, следующий за этим \_\_try. Система и компилятор генерируют дополнительный код для обеспечения выполнения \_\_finally и освобождения локальных объектов в стеке текущего блока. Это относительно дорогостоящая операция.

**Глобальная раскрутка** (global unwind) – это процесс, который происходит при обработке исключения (когда фильтр \_\_except возвращает EXCEPTION\_EXECUTE\_HANDLER). Система выполняет глобальную раскрутку стека от места возникновения исключения до места расположения обработчика исключений. В ходе глобальной раскрутки гарантированно выполняются все незавершенные блоки \_\_finally на пути вверх по стеку вызовов.

1. **В чём главное отличие SEH от С++ exception?**

Главное отличие в контексте программирования заключается в их природе и интеграции с языком. SEH – это механизм операционной системы Windows, который может быть использован из разных языков. Исключения C++ – это часть самого языка C++, и их обработка тесно интегрирована с семантикой языка. В частности, при обработке C++ исключений компилятор C++ автоматически генерирует код для вызова деструкторов объектов (механизм RAII - Resource Acquisition Is Initialization) при раскрутке стека, обеспечивая корректную очистку ресурсов. SEH сам по себе не предоставляет такой автоматической очистки для C++ объектов, хотя компилятор MSVC реализует C++ исключения на основе SEH и использует \_\_finally для реализации вызова деструкторов.

**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

При возникновении исключения ядро ОС перехватывает его и дает возможность коду обработать. Этот механизм называется **Structured Exception Handling** **(SEH)**. Он доступен как для **пользовательского** режима, так и для режима **ядра**. SEH является исключительно частью операционной системы **Windows**, и полная поддержка SEH присутствует только в компиляторе **MSVC.**

Хоть ОС и ловит исключения, основная нагрузка по поддержке SEH лежит на **компиляторе**(а не на ОС). Компилятор:

1. Генерирует специальный код на входах и выходах блоков исключений (exception blocks).
2. Создает таблицы вспомогательных структур.
3. Предоставляет callback-функции (функции обратного вызова, к которым система могла бы обращаться для прохода по блокам исключений) для ОС.
4. Отвечает за стековые фреймы (stack frames) и другую внутреннюю информацию.  **Стековый фрейм** - это область стека для локальных объектов одного блока. Эти понятия важны для раскрутки стека (stack unwinding).

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

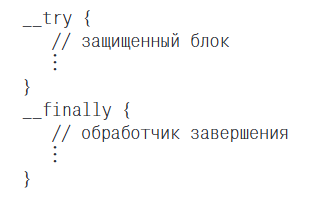
Не путайте SEH с обработкой исключений в C++, которая представляет собой еще одну форму обработки исключений, построенную на применении ключевых слов языка C++ catch и throw. При этом Microsoft Visual C++ использует преимущества поддержки SEH, уже обеспеченной компилятором и операционными системами Windows.

**Cписок ключевых слов SEH, используемых в MSVC:**

* **\_\_try**: Начинает блок, где могут возникнуть исключения.
* **\_\_except**: Указывает, обработано ли исключение, и предоставляет код обработки, если это так.
* **\_\_finally**: Предоставляет код, который гарантированно выполнится независимо от завершения \_\_try блока.
* **\_\_leave**: Оптимизированный механизм перехода к блоку \_\_finally изнутри \_\_try.

# Обработчики завершения - Termination Handling

Обработчик завершения (\_\_finally) гарантирует, что блок кода (собственно обработчик) будет выполнен независимо от того, как происходит выход из другого блока кода – защищенного участка программы. Главная идея - гарантия выполнения кода очистки



Код блока \_\_finally обработчика завершения будет выполнен независимо от того, как произойдет выход из защищенного блока (неважно, разместите Вы в защищенном блоке операторы return или goto).

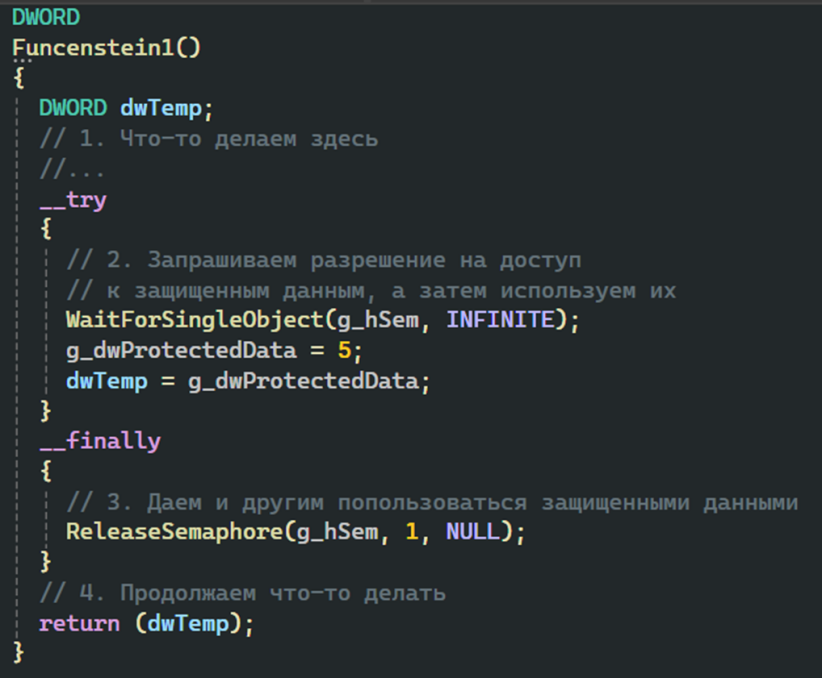
**Защищенный** (охраняемый) блок кода – это блок кода, ограниченный фигурными скобками оператора \_\_try. Предполагается, что в этом блоке может возникнуть исключение, которое следует обработать.

# Пример 1

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.



# Пример 2

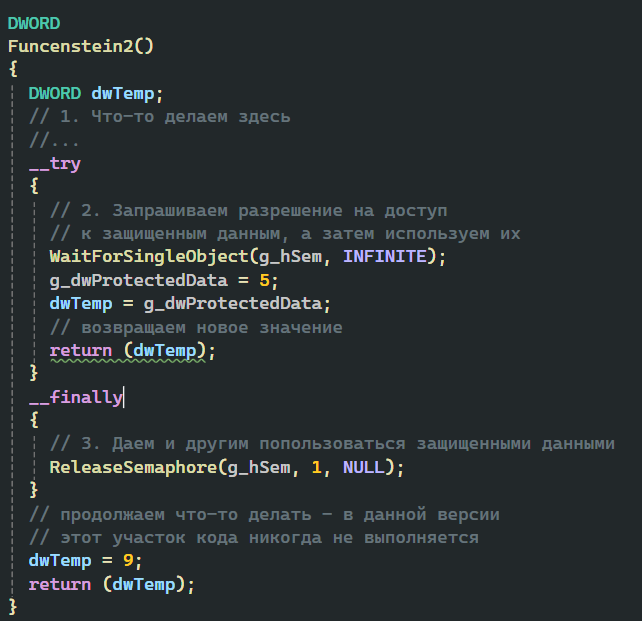
В конец блока \_\_try в функции Funcenstein2 добавлен оператор return. Он сообщает компилятору, что Вы хотите выйти из функции и вернуть значение переменной

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.

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



После выполнения блока \_\_finally функция фактически завершает работу. Любой код за блоком \_\_finally не выполняется, поскольку возврат из функции происходит внутри блока \_\_try

Как компилятор гарантирует выполнение блока \_\_finally до выхода из блока \_\_try:

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)

**Раскрутка** это процесс освобождения локальных объектов каждого из блоков из стека процесса (в частности вложенных блоков).

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

Чтобы все это вытянуть, компилятору приходится генерировать дополнительный код, а системе – выполнять дополнительную работу. Поэтому лучше не писать код, вызывающий преждевременный выход из блока \_\_try обработчика завершения

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.

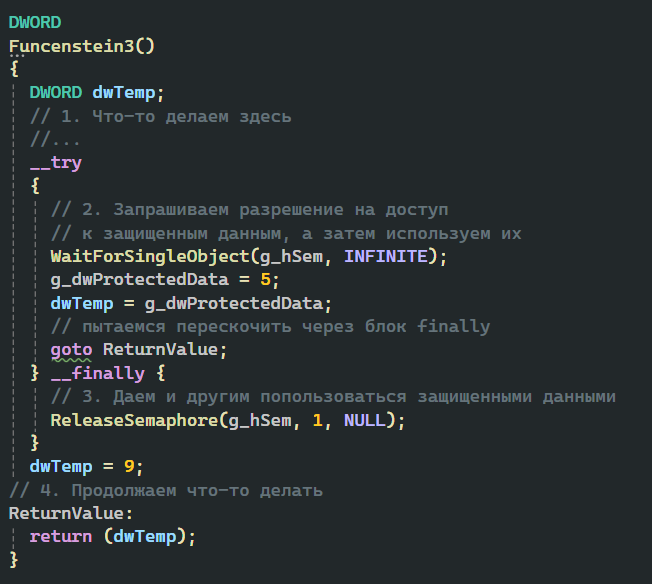
Когда поток управления выходит из блока \_\_try естественным образом (как в Funcenstein1), издержки от вызова блока \_\_finally минимальны, так как для входа в \_\_finally при нормальном выходе из \_\_try исполняется всего одна машинная команда – вряд ли Вы заметите ее влияние на быстродействие своей программы

# Пример 3

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.

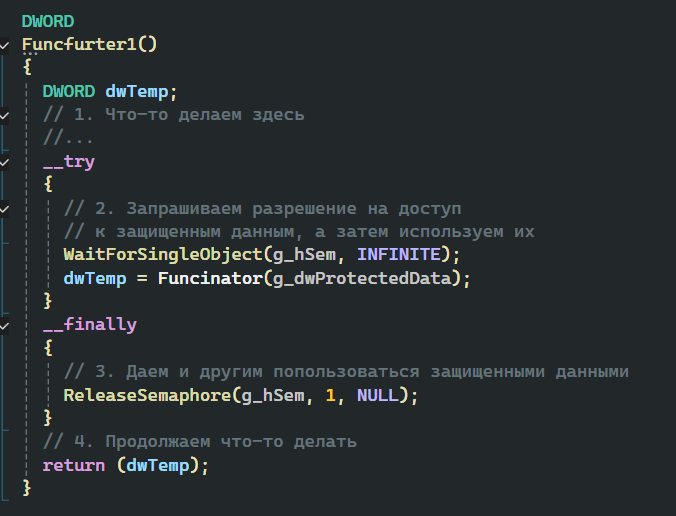
И опять, поскольку Вы прервали естественный ход потока управления из \_\_try в \_\_finally, быстродействие программы – в зависимости от типа процессора – может снизиться весьма значительно



# Пример 4

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.



# Пример 5



Это практический пример для проверки понимания того, как \_\_finally взаимодействует с операторами управления потоком (continue, break) внутри \_\_try в цикле. Нужно проследить выполнение по шагам:

* **dwTemp = 0**. Цикл (0 < 10). \_\_try. Не 2, не 3. \_\_finally (dwTemp становится 1). После \_\_finally (dwTemp становится 2).
* **dwTemp = 2**. Цикл (2 < 10). \_\_try. if (dwTemp == 2) истинно. continue. Выход из \_\_try. Триггерит \_\_finally (dwTemp становится 3).
* **dwTemp = 3**. Цикл (3 < 10). \_\_try. Не 2. if (dwTemp == 3) истинно. break. Выход из \_\_try. Триггерит \_\_finally (dwTemp становится 4).
* **Выход из цикла** (т.к. break). Код после цикла: dwTemp += 10 (dwTemp становится 4 + 10 = 14).
* **return (dwTemp)**возвращает 14.

Вывод: Операторы continue и break внутри \_\_try вызывают выполнение блока \_\_finally.

# Пример 6

Блок \_\_try в Funcenstein4 пытается вернуть значение переменной dwTemp (5) функции, вызвавшей Funcenstein4. 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 (вернёт 103)



# Фильтры и обработчики исключений - Exception Handling

Много проверок ошибок и кода очистки, разбросанного по разным веткам затрудняют чтение, сопровождение и модификацию.

Более структурированные явные проверки ошибок, но с глубокой вложенностью if - легче для понимания, но все еще трудны для модификации/сопровождения из-за отступов и разбросанного кода.

Использование \_\_try-\_\_finally: Вся очистка (освобождение памяти, закрытие файла) собрана в блоке \_\_finally с проверками, были ли ресурсы успешно выделены. Главное достоинство - вся очистка в одном месте, легко добавить новую очистку. Используются return(FALSE) внутри \_\_try.

Проблема - использование return в \_\_try. Ключевое слово \_\_leave решает проблему неоптимальных преждевременных выходов из \_\_try с \_\_finally.

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 (дает минимальные издержки)

1. 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
2. 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

Сценарии, которые приводят к выполнению блока \_\_finally:

1. **Нормальная передача управления** из \_\_try в \_\_finally (через конец \_\_try).
2. **Локальная раскрутка**: преждевременный выход из \_\_try (goto, continue, break, return и т.д.).
3. **Глобальная раскрутка** (global unwind): исключение происходит внутри \_\_try или в вызванной из него функции (как в Funcfurter1). Это также приводит к выполнению \_\_finally в объемлющем блоке.

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.**

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**



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

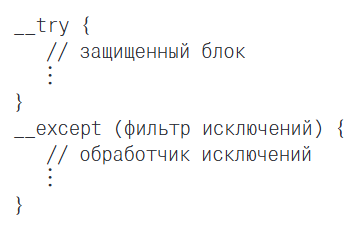
* Упрощается обработка ошибок – очистка гарантируется и проводится в одном месте
* Улучшается восприятие текста программ
* Облегчается сопровождение кода
* Удается добиться минимальных издержек по скорости и размеру кода — при условии правильного применения обработчиков

# Обработчики исключений - Exception Handling

**Исключение** – это событие, которого Вы не ожидали:

* Исключение, возбужденное процессором, называется **аппаратным** (hardware exception)
* Также операционная система и прикладные программы способны возбуждать собственные исключения – **программные** (software exceptions).

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



За блоком \_\_try должен следовать либо \_\_finally, либо \_\_except (нельзя указать оба для одного и того же \_\_try). Нельзя также указывать несколько \_\_finally или \_\_except за одним \_\_try. Однако try-finally можно вкладывать в try-except и наоборот.

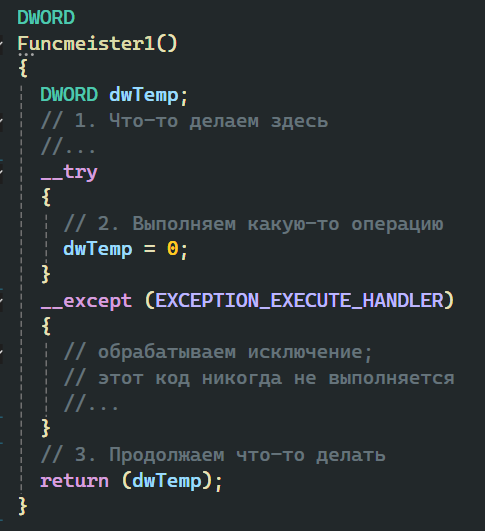
В отличие от обработчиков завершения, фильтры и обработчики исключений выполняются напрямую ОС (нагрузка на компилятор минимальна в момент исключения).

# Пример 7

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



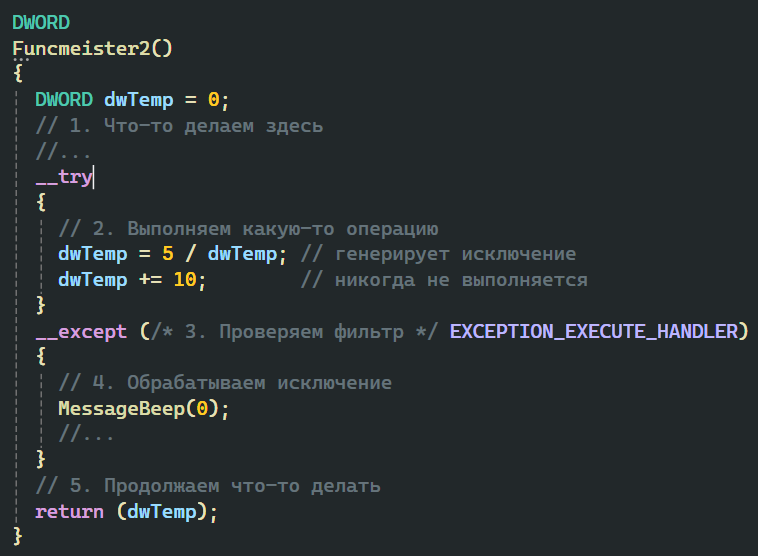
Хотя ставить операторы return, goto, continue и break в блоке \_\_try обработчика завершения настоятельно не рекомендуется, их применение в этом блоке не приводит к снижению быстродействия кода или к увеличению его размера

Использование этих операторов в блоке \_\_try, связанном с блоком \_\_except, не вызовет таких неприятностей, как локальная раскрутка

# Пример 8

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

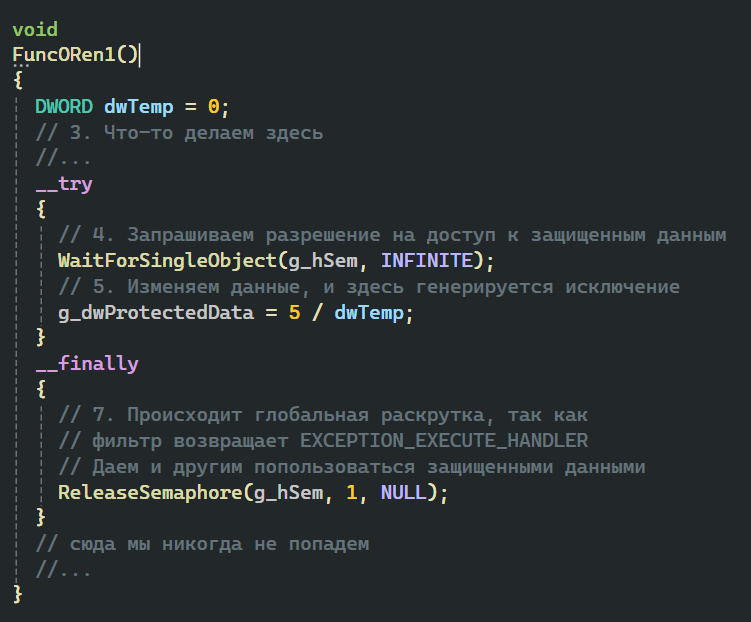
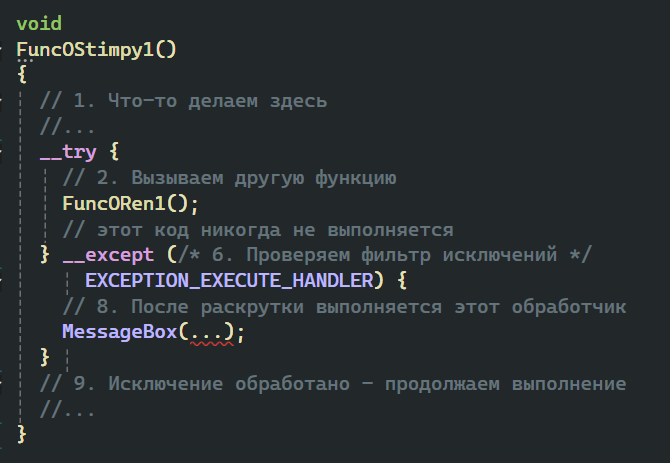


Фильтры исключений:

* 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.

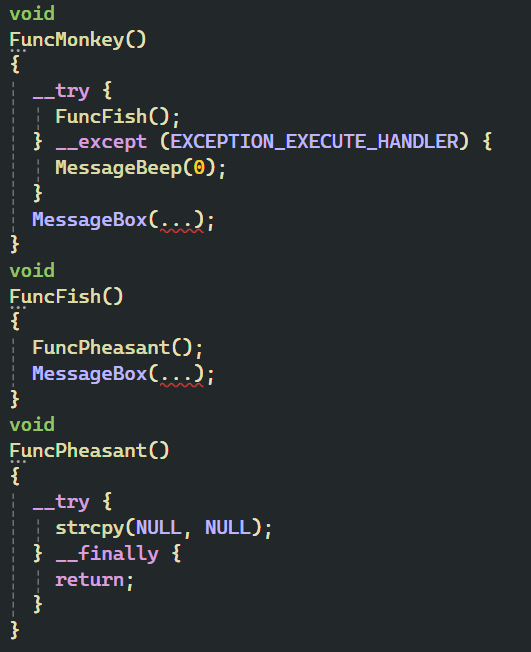


# Пример 9

Глобальную раскрутку, осуществляемую системой, можно остановить, если в блок \_\_finally включить оператор return

Заметьте: код блока \_\_except в FuncMonkey никогда не вызовет MessageBeep.

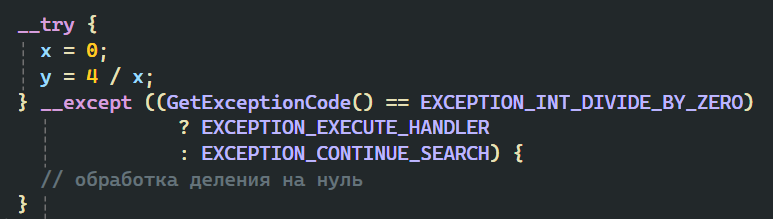
Оператор return в блоке \_\_finally функции FuncPheasant заставит систему вообще прекратить раскрутку, и поэтому выполнение продолжится так, будто ничего не произошло



**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.

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

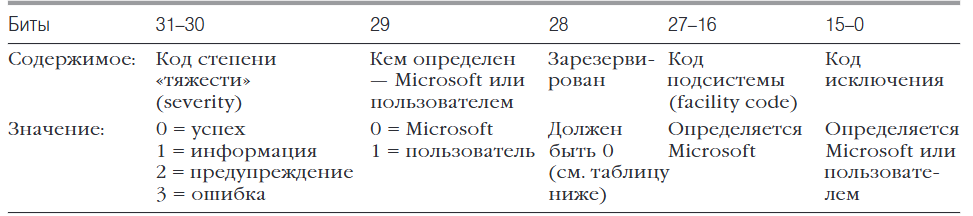


Встраиваемая функция **GetExceptionCode** возвращает **идентификатор типа исключения.**

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

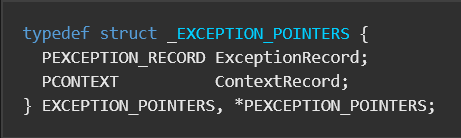
Коды исключений формируются по тем же правилам, что и коды ошибок, определенные в файле WinError.h. Каждое значение типа DWORD разбивается на поля



По сути данная структура исключения соответствует структуре HRESULT из методологии COM.

При возникновении исключения ОС кладет на стек потока три структуры: EXCEPTION\_RECORD, CONTEXT и EXCEPTION\_POINTERS:

1. EXCEPTION\_RECORD содержит аппаратно-независимую информацию об исключении
2. CONTEXT - аппаратно-зависимую.
3. EXCEPTION\_POINTERS содержит указатели на эти две структуры. Показано определение структуры EXCEPTION\_POINTERS.

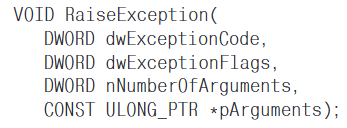


Чтобы получить эту информацию и использовать ее в программе, вызовите **GetExceptionInformation.** Эта встраиваемая функция возвращает указатель на структуру **EXCEPTION\_POINTERS**

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.

Программные исключения можно генерировать самостоятельно. Это альтернатива возвращению особого значения ("признака ошибки") из функции при неудаче. Традиционный подход - функция возвращает код ошибки, а вызывающий код его проверяет.

Альтернативный подход: при неудаче функции возбуждают исключения. Это делает написание и сопровождение кода проще (основной путь выполнения чище) и потенциально быстрее (код проверки ошибок не выполняется в обычном случае). Функция для возбуждения программного исключения: RaiseException.



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.

Собственные программные исключения генерируют в приложениях по целому ряду причин:

1. 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
2. Вполне допустимо возбуждать программные исключения и для уведомления о внутренних фатальных ошибках в приложении

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:



Для изменения стандартного поведения функции UnhandledExceptionFilter можно вызвать функцию:



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:

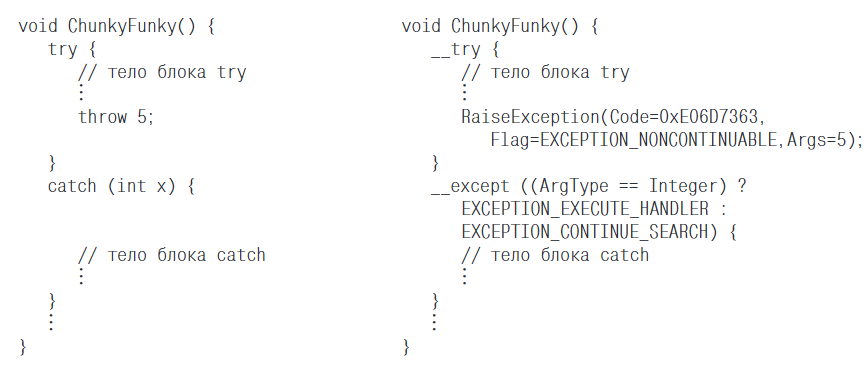


# SEH или исключения C++?

**SEH** – механизм операционной системы, доступный в любом языке программирования, а исключения C++ поддерживаются только в C++

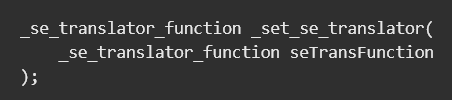
Создавая приложение на C++, Вы должны использовать средства именно этого языка, а не SEH. Причина в том, что исключения C++ – часть самого языка и его компилятор автоматически создает код, который вызывает деструкторы объектов и тем самым обеспечивает корректную очистку ресурсов.

Компилятор MSVC реализует обработку исключений C++ на основе SEH в Windows. C++ try становится SEH \_\_try, C++ catch становится SEH \_\_except (с фильтром, генерируемым компилятором), C++ throw становится вызовом RaiseException (со значением, переданным как аргумент).



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

Это возможно благодаря функции \_set\_se\_translator. Она позволяет установить функцию преобразования SEH-исключений в C++-исключения



**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