**Важно: фильтры вызываются до finally**

**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!

* + Исключение (синхронное): Происходит *непосредственно* в результате выполнения конкретной инструкции в вашем коде. Если вы запустите тот же код с теми же данными, исключение, скорее всего, повторится в том же месте. Пример: деление на ноль.
  + Прерывание (асинхронное): Происходит из-за внешнего события, не связанного напрямую с текущей выполняемой инструкцией. Пример: пользователь нажал клавишу, пришел пакет по сети, таймер сработал. Это может случиться в любой момент выполнения программы.

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

**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.

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

* Visual C++ (MSVC) имеет встроенную поддержку синтаксиса SEH (ключевые слова \_\_try, \_\_except, \_\_finally). Другие компиляторы для Windows (например, GCC/MinGW) могут предоставлять некоторую совместимость через функции WinAPI, но "родная" поддержка на уровне языка – это MSVC.

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. **Ядро само по себе не знает о ваших блоках try, \_\_except, \_\_finally**. Оно знает только, что в некоем потоке некоей программы произошло исключение такого-то типа по такому-то адресу.

* Компилятор (MSVC) делает много скрытой работы:
  + Генерирует специальный код: В начале и в конце ваших \_\_try блоков вставляется дополнительный код.
  + Создает таблицы: Для каждого потока существует цепочка записей об обработчиках исключений. Компилятор генерирует код, который добавляет и удаляет эти записи из цепочки при входе в \_\_try блок и выходе из него.
  + Функции обратного вызова (callbacks): Эти таблицы, по сути, содержат адреса функций (фильтров и обработчиков), которые ОС будет вызывать при поиске обработчика исключения.

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

* Стековые фреймы (Stack Frames):
  + Когда вызывается функция, в стеке для нее выделяется место. Эта область называется стековым фреймом. В ней хранятся:
    - Локальные переменные функции.
    - Аргументы, переданные функции (в некоторых соглашениях о вызовах).
    - Адрес возврата (куда вернуться после завершения функции).
    - Сохраненные значения регистров.
  + Компилятор управляет созданием и уничтожением этих фреймов.
* Раскрутка стека (Stack Unwinding): Это критически важный процесс.
  + Представьте цепочку вызовов: main() -> funcA() -> funcB() -> funcC().
  + Если в funcC() происходит исключение, и funcC() его не обрабатывает, ОС начинает "раскручивать стек".
  + Она смотрит: есть ли обработчик в funcC()? Нет.
  + "Уничтожает" стековый фрейм funcC(). При этом, если это C++ код, должны вызваться деструкторы локальных объектов funcC(). Для SEH с \_\_finally, должен выполниться блок \_\_finally funcC().
  + Переходит к funcB(). Есть ли обработчик в funcB()? Если да, и он готов обработать, то он выполняется. Если нет, то фрейм funcB() тоже уничтожается (с выполнением \_\_finally или деструкторов).
  + И так далее, вверх по стеку вызовов, пока не найдется обработчик или пока не дойдем до самого верха (тогда приложение обычно падает).
  + Важно для SEH: \_\_finally блоки выполняются во время этой раскрутки стека, даже если исключение не будет обработано в той же функции. Это гарантирует освобождение ресурсов.

Две возможности SEH:

* Обработка завершения (Termination Handling):
  1. Ключевое слово: \_\_finally.
  2. Цель: Гарантировать выполнение определенного блока кода (блока \_\_finally) *независимо от того, как завершился предыдущий блок кода (\_\_try)*. Это может быть нормальное завершение, выход через return, goto, или даже если произошло исключение.
  3. Использование: Освобождение ресурсов (закрытие файлов, мьютексов, освобождение памяти).
* Обработка исключений (Exception Handling):
  1. Ключевое слово: \_\_except.
  2. Цель: Перехватить и обработать ошибку (исключение), возникшую в блоке \_\_try.
  3. Использование: Реакция на конкретные ошибки, логирование, попытка восстановления.

SEH vs. C++ Исключения (try/catch/throw):

* SEH: Механизм операционной системы Windows, доступный на более низком уровне. Ключевые слова \_\_try, \_\_except, \_\_finally являются расширениями языка C/C++ в компиляторе MSVC. SEH может перехватывать *любые* исключения, включая аппаратные (деление на ноль, ошибка доступа к памяти). По умолчанию SEH *не вызывает* деструкторы C++ объектов автоматически при раскрутке стека (есть нюансы, но общее правило такое).
* C++ Исключения: Стандартный механизм языка C++. Использует try, catch, throw. Предназначен для обработки ошибок на уровне логики программы. Гарантирует вызов деструкторов локальных объектов при раскрутке стека (принцип RAII). C++ catch по умолчанию не ловит аппаратные SEH-исключения (например, ошибку доступа к памяти), если только не используется специальный транслятор (об этом позже).
* Связь в MSVC: Компилятор MSVC реализует C++ исключения *поверх* SEH. Когда вы пишете throw в C++, MSVC генерирует вызов функции RaiseException (которая является частью SEH). Когда вы пишете catch, MSVC генерирует соответствующий SEH-фильтр. Это позволяет C++ коду в MSVC быть более интегрированным с низкоуровневыми ошибками Windows, но также добавляет специфику.

\_\_try / Начинает блок кода, в котором могут возникать исключения. Код внутри него находится под "наблюдением" SEH. Называется защищённым/охраняемым блоком.

\_\_except (выражение\_фильтра) { блок\_обработчика }:

* Используется для *обработки ошибок*.
* выражение\_фильтра – это самое важное. Оно вычисляется *только если* в \_\_try произошло исключение. Результат этого выражения (одно из трех специальных значений) говорит системе, что делать дальше.
* блок\_обработчика – выполняется, если фильтр решил обработать исключение.

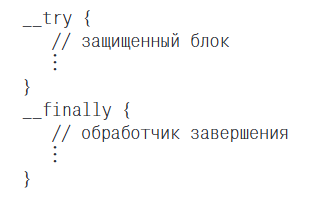
\_\_finally / Предоставляет код, который гарантированно будет выполнен независимо от того, завершается ли блок \_\_try обычным образом, с помощью инструкции return или из-за исключения

\_\_leave / Предоставляет оптимизированный механизм для перехода к блоку \_\_finally откуда-либо из блока \_\_try (на самом деле, \_\_leave переходит к концу блока \_\_try, после чего выполняется \_\_finally)

Важное правило: Блок \_\_try может сопровождаться *либо* \_\_except, *либо* \_\_finally, но не обоими одновременно для одного и того же \_\_try. Если нужны оба, их нужно вкладывать друг в друга (например, \_\_try { \_\_try { ... } \_\_except { ... } } \_\_finally { ... }).

# \_finally

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:



Если вы внутри \_\_try напишете return; или goto some\_label\_outside;, компилятор сгенерирует специальный код. Перед тем как return или goto фактически выполнятся, будет выполнен блок \_\_finally. Это называется "локальная раскрутка" (local unwind).

**Локальная раскрутка (local unwind)** в контексте SEH (Structured Exception Handling) — это процесс, при котором компилятор гарантирует выполнение кода из блока \_\_finally **перед тем**, как управление фактически покинет ассоциированный с ним блок \_\_try из-за инструкций **преждевременного** выхода.

# "Локальная раскрутка" (Local Unwind)

Компилятор MSVC достаточно умен. Когда он видит return (или goto, break, continue) внутри \_\_try, который имеет соответствующий \_\_finally, он генерирует специальный код:

1. Значение, которое должно быть возвращено, временно сохраняется.
2. Выполняется код из блока \_\_finally.
3. После завершения \_\_finally, происходит фактический возврат сохраненного значения.
4. Этот процесс называется локальной раскруткой (local unwind). Это гарантирует, что "уборка" в \_\_finally всегда происходит.

По сути раскрутка это процесс освобождения локальных объектов каждого из блоков из стека процесса (в частности вложенных блоков)

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

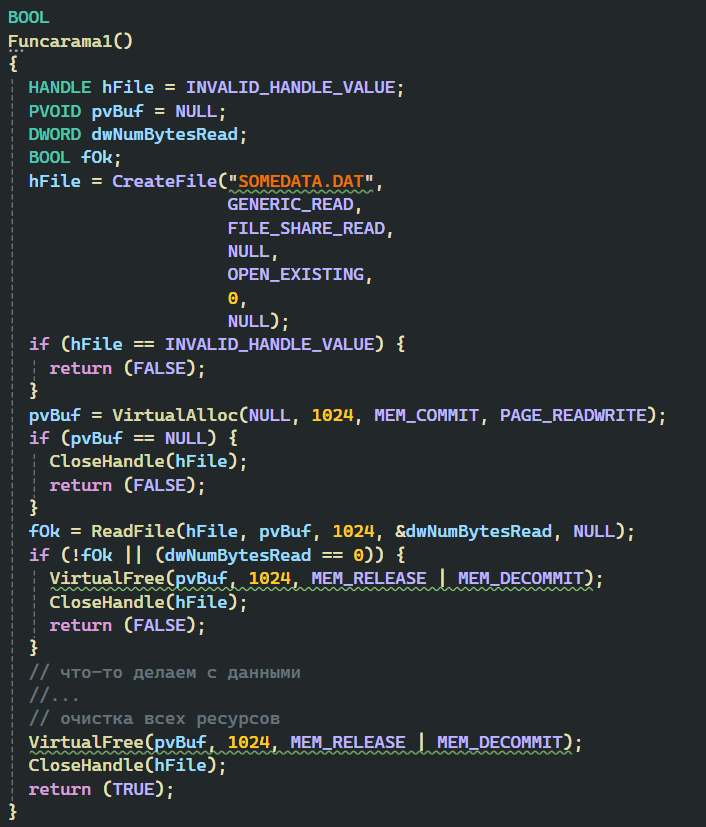
SEH (особенно \_\_except) не предназначена для управления обычным потоком выполнения программы. Если какая-то "ошибка" происходит часто и является ожидаемой частью логики, лучше использовать обычные if/else проверки. Например, проверка, что пользователь ввел число, а не буквы.

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

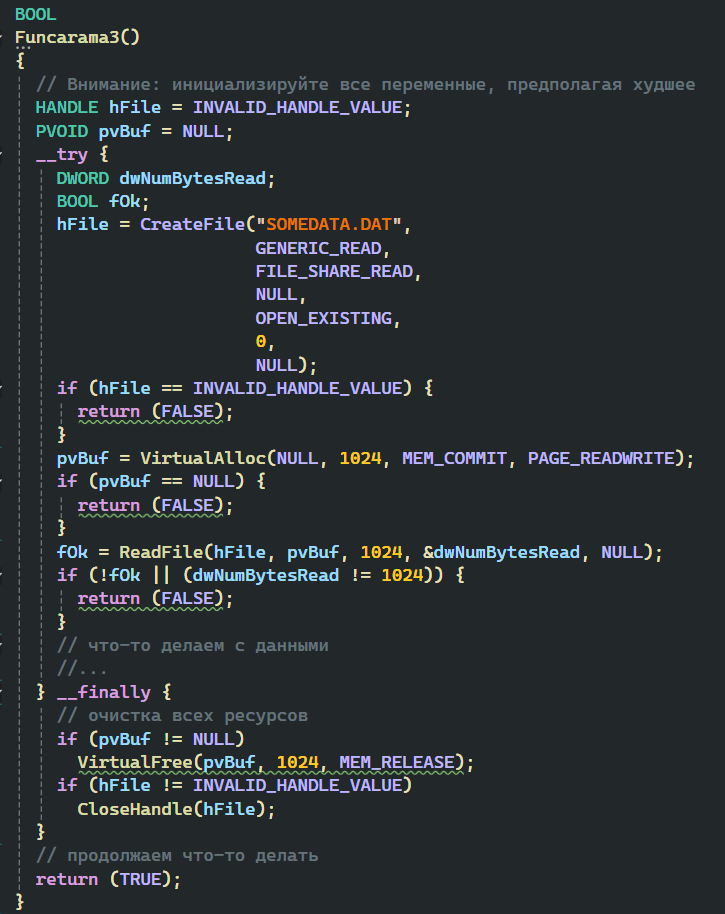
Заметьте: когда поток управления выходит из блока \_\_try естественным образом, издержки от вызова блока finally минимальны, так как для входа в \_\_finally при нормальном выходе из \_\_try исполняется всего одна машинная команда – вряд ли Вы заметите ее влияние на быстродействие своей программы

\_\_finally работает не только для return/goto из \_\_try, но и для исключений, произошедших в \_\_try или в функциях, вызванных из \_\_try. Он обеспечивает надежную очистку.

Если return есть и в \_\_try, и в \_\_finally, то return из \_\_finally "**перебивает**" return из \_\_try. Функция немедленно завершается, возвращая значение из \_\_finally.

  
Это классический пример, который возникает при ручной обработке ошибок и освобождении ресурсов.

* **Дублирование кода очистки:** CloseHandle(hFile) вызывается в нескольких местах (A). VirtualFree(...) тоже может дублироваться (B).
* **Трудно отследить:** При каждом return нужно убедиться, что все *ранее захваченные* ресурсы освобождены. Если добавить новый ресурс посередине, придется обновить все последующие пути выхода.
* **Легко ошибиться:** Пропустить освобождение ресурса в одном из путей выхода – частая ошибка, приводящая к утечкам.



* **Централизованная очистка:** Весь код освобождения ресурсов (VirtualFree, CloseHandle) находится в одном блоке \_\_finally.
* **Инициализация "плохими" значениями:** hFile = INVALID\_HANDLE\_VALUE; pvBuf = NULL; Это важно. В \_\_finally мы проверяем, были ли ресурсы успешно выделены, прежде чем пытаться их освободить.
* **Досрочные return (FALSE);:** Если какая-то операция в \_\_try не удалась, мы можем просто сделать return (FALSE);. Блок \_\_finally **все равно будет выполнен** перед фактическим выходом из функции, обеспечивая очистку.
* **Проблема return в \_\_try:** Как уже обсуждалось, return из \_\_try (когда есть \_\_finally) вызывает локальную раскрутку, что имеет небольшие накладные расходы. Также, если функция должна возвращать TRUE при успехе, а все return в \_\_try – это return FALSE, то нужен return TRUE где-то после \_\_try/\_\_finally или установка флага.

**\_\_leave** – это способ выйти из \_\_try блока, который сопряжен с \_\_finally, без накладных расходов return или goto на локальную раскрутку, *но с гарантированным выполнением \_\_finally*.

* **Как работает \_\_leave:**
  1. Когда встречается \_\_leave; внутри \_\_try.
  2. Управление немедленно передается *в* ***конец* блока \_\_try**.
  3. Затем, как обычно, выполняется блок \_\_finally.
  4. После выполнения \_\_finally, выполнение продолжается с инструкции, следующей за блоком \_\_finally.
* Поскольку \_\_leave не возвращает значение, нам нужен флаг (здесь fFunctionOk), чтобы запомнить, успешно ли завершился \_\_try. Этот флаг устанавливается в TRUE только если все операции в \_\_try прошли успешно. Функция возвращает значение этого флага.
* **Преимущества Funcarama4:**
  1. Централизованная очистка в \_\_finally.
  2. Нет return внутри \_\_try, что избегает более "дорогой" локальной раскрутки. Переход к \_\_finally через \_\_leave более эффективен.
  3. Код в \_\_try становится чище, так как он фокусируется на логике и установке флага успеха/неуспеха (через \_\_leave).

\_\_try { // TF\_Outer

\_\_try { // TF\_Inner

if (Echo(Counter) == Counter) {

Counter += 3;

\_\_leave;

} else {

Counter += 100;

}

} \_\_finally {

if (abnormal\_termination() == FALSE) {

Counter += 5;

\_\_leave;

}

}

Counter += 100;

} \_\_finally {

if (abnormal\_termination() == FALSE) {

Counter += 5;

}

}

**Ключевой момент!** Этот \_\_leave находится внутри \_\_finally TF\_Inner. Оператор \_\_leave всегда относится к ближайшему охватывающему его \_\_try блоку. В данном случае, \_\_finally TF\_Inner сам по себе не является \_\_try блоком. Ближайший \_\_try блок, который "охватывает" это место — это \_\_try TF\_Outer. Таким образом, этот \_\_leave инициирует выход из \_\_try блока TF\_Outer.

**Слайд 27: Рекомендации по разработке с \_\_finally**

* 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

|  |  |
| --- | --- |
| **Локальная раскрутка** | **Глобальная раскрутка** |
| Это процесс, при котором система гарантирует выполнение кода из блока \_\_finally, когда управление **преждевременно покидает** блок \_\_try **без возникновения исключения**, но путем использования ненормальных путей выхода.  Компилятор вставляет специальный код, который перед фактическим выполнением return, goto или \_\_leave сначала выполняет содержимое \_\_finally. **Это происходит в рамках текущего кадра стека (функции, где находится \_\_try).**  Локальная раскрутка **не связана с активным исключением**. Она происходит при нормальном (хотя и преждевременном) потоке управления. | Это процесс, который происходит, когда **исключение действительно возникает** (аппаратное или программное) и система начинает искать подходящий обработчик для этого исключения. В этом процессе происходит **пошаговое "разматывание" (unwinding) стека вызовов** от точки возникновения исключения до того места, где найден обработчик.  Для каждого кадра стека, который "разматывается", выполняются следующие действия:   * Если это кадр с активными C++ объектами, вызываются их деструкторы. * Если это кадр с блоком \_\_try и соответствующим \_\_finally, код из \_\_finally выполняется. |
| * return внутри \_\_try. * goto из \_\_try на метку за его пределами. * break или continue из цикла, который охватывает \_\_try. * Использование инструкции \_\_leave. | * Аппаратные исключения (деление на ноль, нарушение доступа к памяти). * Программные исключения (например, RaiseException в Windows SEH или throw в C++). |

**ВОПРОС: в каком порядке выполняются finally при глобальной раскрутке?**

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

* **нормальная передача управления от блока \_\_try блоку \_\_finally** (код в \_\_try дошел до конца)
* **локальная раскрутка –** преждевременный выход из блока \_\_try (из-за операторов goto, continue, break, return и т. д.), вызывающий принудительную передачу управления блоку \_\_finally
* **Глобальная раскрутка (Global Unwind):** Внутри \_\_try (или в функции, вызванной из \_\_try) произошло *исключение* (не return/goto, а именно SEH-исключение, например, access violation). Система начинает раскрутку стека в поисках обработчика \_\_except. На пути раскрутки, если она проходит через функцию, содержащую \_\_try/\_\_finally, то этот \_\_finally будет выполнен. Это происходит *до* того, как будет найден и выполнен соответствующий \_\_except (если он вообще найдется).

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. Если управление естественным образом передано из \_\_try в \_\_finally, AbnormalTermination возвращает FALSE. А если выход был преждевременным – то вызов **AbnormalTermination** дает TRUE

**Зачем это нужно?** Иногда в \_\_finally нужно выполнить разные действия в зависимости от того, как завершился \_\_try. Например, если \_\_try завершился нормально, можно залогировать успех. Если произошла ошибка (преждевременный выход), можно залогировать ошибку или выполнить дополнительную очистку.

**Leave не считается аномальным выходом.**

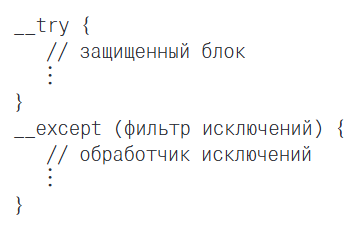
**Аппаратные исключения (Hardware Exceptions):** Генерируются непосредственно процессором (CPU).

* Примеры:
  + Деление на ноль (DIVIDED\_BY\_ZERO).
  + Нарушение доступа к памяти (Access Violation - ACCESS\_VIOLATION): попытка чтения/записи по адресу, к которому нет доступа, или который вообще не существует.
  + Недопустимая инструкция (ILLEGAL\_INSTRUCTION): процессор не понимает код, который ему дали на исполнение.
  + Переполнение стека (STACK\_OVERFLOW): хотя это часто проявляется как access violation, когда запись выходит за пределы стека.
  + Точка останова (BREAKPOINT): инструкция INT 3.

**Программные исключения (Software Exceptions):** Генерируются не процессором, а кодом – либо самой операционной системой, либо вашим собственным приложением (или библиотекой).

* ОС может генерировать программные исключения для сигнализации об определенных системных ошибках.
* Ваше приложение может генерировать программные исключения с помощью функции RaiseException(). Это способ для одной части программы сигнализировать другой части (или системе в целом) о серьезной ошибке.
* C++ throw в MSVC под капотом использует RaiseException(), поэтому C++ исключения являются видом программных исключений.
* **Важно:** SEH (\_\_try/\_\_except) может ловить *оба* типа этих исключений.

**Синтаксис обработчика исключений таков:**



**\_\_try { ... }**: Тот же самый "защищенный блок", что и с \_\_finally. Если внутри этого кода происходит исключение, система начинает искать обработчик.

**\_\_except (выражение\_фильтра) { ... }**: Этот блок выполняется *только если* в \_\_try произошло исключение *и* выражение\_фильтра решило, что это исключение нужно обработать здесь.

* **выражение\_фильтра**: Это сердце \_\_try/\_\_except. Это выражение вычисляется *в контексте возникновения исключения*. Оно может вызывать специальные функции (например, GetExceptionCode(), GetExceptionInformation()) для получения деталей об ошибке.

**За блоком \_\_try всегда должен следовать либо блок \_\_finally, либо блок \_\_except**. **Для блока \_\_try нельзя указать одновременно и блок \_\_finally, и блок \_\_except; к тому же за \_\_try не может следовать несколько блоков \_\_finally или \_\_except** (одно \_\_try - одно \_\_finally ИЛИ одно \_\_except).

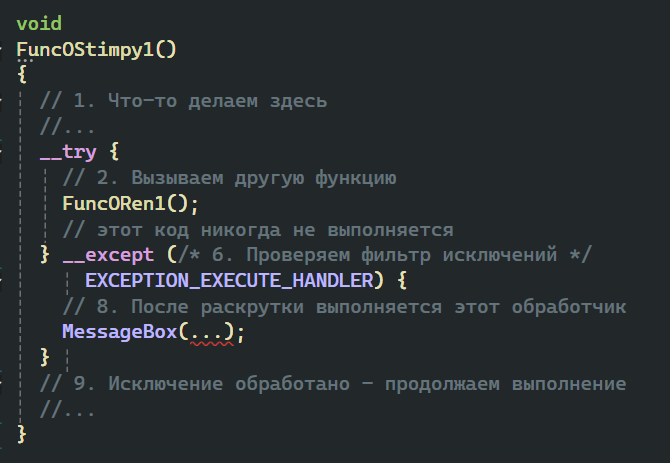
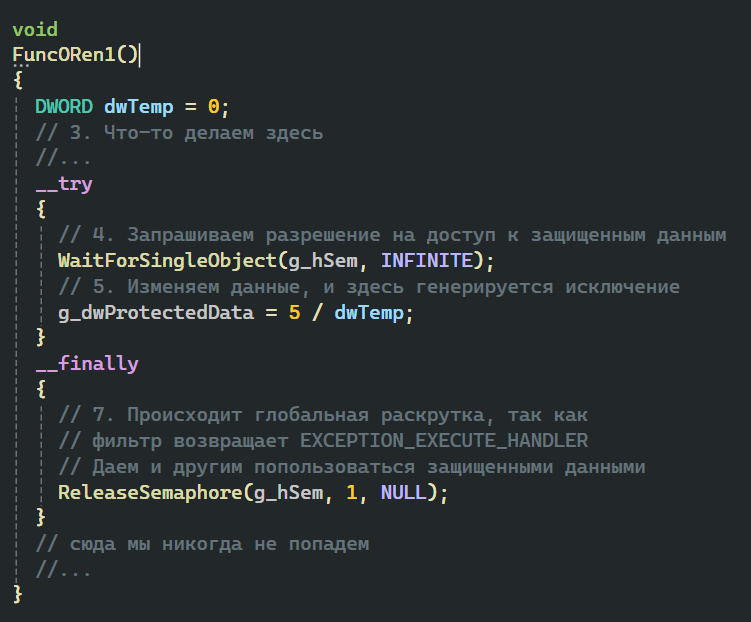
**Однако try-finally можно вложить в try-except, и наоборот**

**В отличие от обработчиков завершения, фильтры и обработчики исключений выполняются непосредственно операционной системой – нагрузка на компилятор при этом минимальна** (по сравнению с генерацией кода для локальной раскрутки \_\_finally).

Использование return/goto и т.п. в \_\_try, за которым следует \_\_except, не создает дополнительных накладных расходов, *если эти return/goto выполняются до возникновения какого-либо исключения*. Если же исключение происходит, то эти return/goto просто не будут достигнуты.

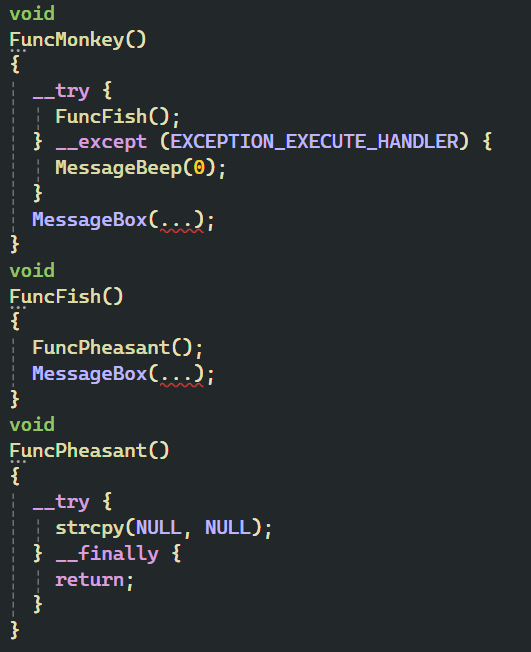
**Фильтры исключений**

* **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 до уровня этого try-except блока**, а затем управление передается коду внутри блока \_\_except (коду обработчика исключений). После его выполнения система считает исключение обработанным и разрешает программе продолжить работу
* **Глобальная раскрутка:** Прежде чем выполнить код в \_\_except { ... }, система должна "очистить" стек до текущего уровня. Это означает, что если исключение произошло в глубоко вложенной функции (main -> A -> B -> C!bang), и обработчик находится в main, то стековые фреймы функций C, B, A будут уничтожены. При этом, если в C, B, или A были \_\_try/\_\_finally блоки, их \_\_finally части будут выполнены во время этой раскрутки. Это гарантирует освобождение ресурсов даже при "прыжке" через несколько уровней стека к обработчику.
* **Продолжение работы:** После \_\_except { ... }, программа продолжает выполняться с инструкции, следующей за этим блоком \_\_except. Исключение считается "погашенным".

1. FuncOStimpy1 начинает выполняться (Шаг 1).
2. Внутри \_\_try блока FuncOStimpy1, вызывается Func0Ren1() (Шаг 2).
3. Func0Ren1 начинает выполняться (Шаг 3).
4. Func0Ren1 входит в свой \_\_try блок (Шаг 4).
5. В \_\_try блока Func0Ren1 происходит деление на ноль: g\_dwProtectedData = 5 / dwTemp; (Шаг 5). **Исключение!**
6. ОС начинает поиск обработчика.
   * В Func0Ren1 нет \_\_except, только \_\_finally. ОС "пропускает" Func0Ren1 в поиске \_\_except.
   * ОС "возвращается" в FuncOStimpy1. Там есть \_\_try/\_\_except.
   * Вычисляется фильтр \_\_except в FuncOStimpy1. В примере это EXCEPTION\_EXECUTE\_HANDLER (Шаг 6). Это означает, что FuncOStimpy1 будет обрабатывать это исключение.
7. **Начинается глобальная раскрутка стека** от точки исключения в Func0Ren1 до \_\_try/\_\_except блока в FuncOStimpy1.
   * Во время этой раскрутки, ОС видит, что функция Func0Ren1 (чей стековый фрейм сейчас уничтожается) имеет \_\_try/\_\_finally. Поэтому **блок \_\_finally в Func0Ren1 выполняется** (Шаг 7). OutputDebugString("Func0Ren1: In \_\_finally\n"); будет выведено. (Если бы был семафор, он бы освободился).
8. После того как раскрутка стека завершена до FuncOStimpy1, выполняется код в блоке \_\_except функции FuncOStimpy1. MessageBox будет показан (Шаг 8).
9. Исключение считается обработанным. Выполнение FuncOStimpy1 продолжается с кода, следующего за блоком \_\_except (Шаг 9).

**Ключевой момент:** \_\_finally в Func0Ren1 выполняется *во время* раскрутки стека, *перед* тем, как \_\_except в FuncOStimpy1 получит управление. Это гарантирует очистку ресурсов.

****

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

**Оператор return в блоке \_\_finally функции FuncPheasant заставит систему вообще прекратить раскрутку, и поэтому выполнение продолжится так, будто ничего не произошло** (имеется в виду, что исключение "поглощается" и не доходит до \_\_except в FuncMonkey).

**Комментарий к этому очень важному и коварному примеру:**

1. FuncMonkey вызывает FuncFish.
2. FuncFish вызывает FuncPheasant.
3. В FuncPheasant, в блоке \_\_try, strcpy(NULL, NULL) вызывает исключение (Access Violation).
4. **Начинается поиск обработчика (первый проход – поиск фильтра):**
   * ОС смотрит на текущую функцию FuncPheasant. В ней нет \_\_except блока.
   * ОС "поднимается" по стеку к FuncFish. В FuncFish тоже нет \_\_except блока.
   * ОС "поднимается" по стеку к FuncMonkey. В FuncMonkey есть \_\_try/\_\_except блок.
   * **Вызывается фильтр \_\_except блока в FuncMonkey**. В данном случае, это константа EXCEPTION\_EXECUTE\_HANDLER. Этот фильтр говорит: "Да, я буду обрабатывать это исключение".
5. **Теперь, когда обработчик найден (фильтр вернул EXCEPTION\_EXECUTE\_HANDLER), начинается вторая фаза – глобальная раскрутка стека (global unwind):**
   * Система начинает "разматывать" стек от точки возникновения исключения (FuncPheasant) до уровня найденного обработчика (FuncMonkey).
   * **Раскрутка фрейма FuncPheasant:**
     1. ОС видит, что FuncPheasant содержит \_\_try/\_\_finally.
     2. **Выполняется блок \_\_finally в FuncPheasant.**
     3. Внутри этого \_\_finally находится return;.
   * **Эффект return из \_\_finally во время глобальной раскрутки:**
     1. Этот return; немедленно завершает выполнение функции FuncPheasant.
     2. **Критически важно:** Он также **прерывает (останавливает) процесс глобальной раскрутки стека!** Система считает, что исключение было "обработано" этим return (хотя это не классическая обработка через \_\_except). Раскрутка не продолжается до FuncFish или FuncMonkey для выполнения \_\_except блока.
6. Поскольку глобальная раскрутка была остановлена оператором return в \_\_finally функции FuncPheasant, управление возвращается из FuncPheasant в FuncFish так, как будто FuncPheasant просто нормально завершилась (хотя и из \_\_finally). Исключение "погашено".
7. FuncFish продолжает свое выполнение с точки после вызова FuncPheasant. MessageBox("FuncFish continues...", ...) будет показан. (В коде на слайде этот MessageBox отсутствует, но если бы был, он бы выполнился).
8. FuncFish завершается и возвращает управление в FuncMonkey.
9. Для FuncMonkey вызов FuncFish() завершился *нормально* (потому что исключение было "поглощено" в FuncPheasant). Поэтому блок \_\_try в FuncMonkey считается успешно завершенным.
10. Следовательно, фильтр и тело блока \_\_except в FuncMonkey **не выполняются**. MessageBeep(0) не будет.
11. FuncMonkey продолжает выполнение после \_\_except блока. MessageBox("FuncMonkey continues...", ...) будет показан.

Если бы у FuncFish был свой \_\_finally, то выполнился бы он или нет, зависело бы от того, остановила ли FuncPheasant раскрутку *до* того, как раскрутка "дошла" бы до фрейма FuncFish. В данном случае, return в \_\_finally FuncPheasant происходит немедленно, так что раскрутка не дойдет до обработки \_\_finally в FuncFish, если бы он там был и если бы исключение произошло ниже FuncFish.

**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**

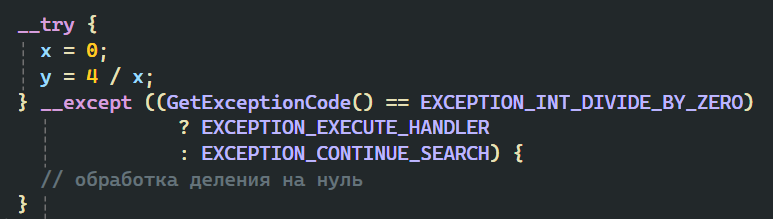
**Комментарий:**

1. **EXCEPTION\_CONTINUE\_EXECUTION (числовое значение -1):**

* **Код внутри тела блока \_\_except { ... } НЕ выполняется.** Этот блок предназначен для выполнения, только если фильтр возвращает EXCEPTION\_EXECUTE\_HANDLER.
* **Раскрутка стека НЕ происходит.** Система не "разматывает" стек до этого \_\_try/\_\_except блока.
* **Система пытается повторно выполнить ту самую инструкцию, которая вызвала исключение.** Она предполагает, что условия изменились (например, вы в фильтре каким-то образом исправили причину ошибки), и теперь инструкция может выполниться успешно.

**Ключевая идея:** EXCEPTION\_CONTINUE\_EXECUTION говорит системе: "Я считаю, что я исправил(а) проблему прямо в коде фильтра. Пожалуйста, попробуй еще раз выполнить ту же самую машинную инструкцию, которая только что упала".

1. **EXCEPTION\_CONTINUE\_SEARCH (числовое значение 0):**
   * **Предназначение:** Это стандартный способ сказать: "Это не мое исключение, или я не хочу его здесь обрабатывать. Пожалуйста, поищите другой обработчик (\_\_except) выше по стеку вызовов."
   * **Поиск обработчика:** ОС продолжает раскрутку стека, ищет следующий \_\_try/\_\_except блок. Если находит, вызывает его фильтр. Если доходит до самого верха стека и не находит обработчик, который бы вернул EXCEPTION\_EXECUTE\_HANDLER, то это становится **необработанным исключением** (unhandled exception), и система обычно завершает программу (часто с показом диалога об ошибке).



Часто фильтр исключений должен проанализировать ситуацию, прежде чем определить, какое значение ему вернуть.

**GetExceptionCode():** Это встроенная функция, которую можно вызывать ***внутри выражения-фильтра \_\_except* или *внутри тела блока \_\_except***. Она возвращает код исключения (например, EXCEPTION\_INT\_DIVIDE\_BY\_ZERO, EXCEPTION\_ACCESS\_VIOLATION).

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

**Однако GetExceptionCode нельзя вызывать из обычной функции, которая вызывается из фильтра.**

* **Причина ограничения:** GetExceptionCode() (и GetExceptionInformation()) работают с информацией об исключении, которую система подготавливает *специально* для контекста выполнения фильтра или обработчика. Вне этого контекста эта информация недоступна или некорректна.

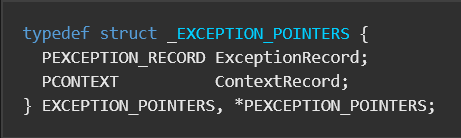
**Коды исключений** формируются по тем же правилам, что и коды ошибок, определенные в файле WinError.h. Каждое значение типа DWORD разбивается на поля.

* + **Биты 31-30: Severity (тяжесть)**
    - 00 (0) = Success (для кодов ошибок, не для исключений обычно)
    - 01 (1) = Informational
    - 10 (2) = Warning
    - 11 (3) = Error (исключения обычно имеют эту тяжесть)
  + **Бит 29: Customer code flag.** Позволяет различать коды, определенные Microsoft, и коды, определенные пользовательскими приложениями
    - 0 = Microsoft defined
    - 1 = Customer defined (т.е. определено приложением/разработчиком)
  + **Бит 28: Reserved (должен быть 0)**
  + **Биты 27-16: Facility code (код подсистемы/компонента).** Указывает на "источник" ошибки.
    - Определяет, какая часть системы или приложения сгенерировала код.
  + **Биты 15-0: Code (собственно код ошибки/исключения в рамках facility)**
* **По сути данная структура исключения соответствует структуре HRESULT из методологии СОМ**

**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 (Содержит состояние регистров процессора **на момент возникновения исключения**) – машинно-зависимую информацию об этом исключении.

В структуре EXCEPTION\_POINTERS всего два элемента – указатели на помещенные в стек структуры EXCEPTION\_RECORD и CONTEXT



Эту структуру возвращает функция **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**.

**Почему только в фильтре?** Информация, на которую указывают EXCEPTION\_POINTERS (сами EXCEPTION\_RECORD и CONTEXT), размещается системой на стеке *временно*, на время работы механизма поиска обработчика. Как только фильтр отработал и управление передано (или не передано) в тело \_\_except { ... }, эти стековые данные могут быть перезаписаны.

# Дальше про программные исключения

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

**Проблема:** Вызывающий код *обязан* проверять возвращаемое значение после каждого вызова.

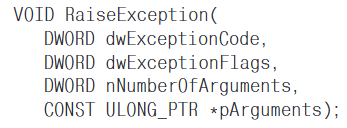
Легко забыть проверить код возврата, что может привести к работе с некорректными данными или состоянием.

**Альтернативный подход (программные исключения):**

Если функция сталкивается с ошибкой, она не возвращает специальное значение, а "бросает" (генерирует) исключение.

Вызывающий код может находиться внутри блока \_\_try (для SEH) или try (для C++ исключений), и соответствующий \_\_except или catch блок может перехватить это исключение.

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



1. Ее первый параметр, dwExceptionCode, – значение, которое идентифицирует генерируемое исключение

* + Это код вашего программного исключения. Вы можете использовать один из стандартных кодов ошибок Windows (если это уместно) или определить свой собственный.
  + Если вы определяете свой собственный код, убедитесь, что он соответствует формату, описанному на слайде 45 (severity, customer flag, facility, code). Например, можно использовать старшие биты 0xE0000000 (как C++ исключения) или установить customer flag.
  + Этот код будет доступен через GetExceptionCode() или pExceptionInfo->ExceptionRecord->ExceptionCode в фильтре/обработчике.

2. 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. EXCEPTION\_NONCONTINUABLE: Указывает, что это исключение не может быть продолжено. Если фильтр попытается вернуть EXCEPTION\_CONTINUE\_EXECUTION для такого исключения, система сгенерирует новое исключение EXCEPTION\_INVALID\_DISPOSITION (неверное решение). Это флаг "серьезности" – ошибка настолько плоха, что пытаться продолжить бессмысленно.

**3. nNumberOfArguments (DWORD):**

* + Количество элементов в массиве pArguments.
  + Максимум EXCEPTION\_MAXIMUM\_PARAMETERS (обычно 15). Если передать больше, лишние будут проигнорированы.

4. \**pArguments (CONST ULONG\_PTR ):*

* + Указатель на массив значений типа ULONG\_PTR (это беззнаковое целое, размер которого равен размеру указателя, т.е. 32 бита на x86, 64 бита на x64).
  + Эти аргументы будут скопированы в поле ExceptionInformation структуры EXCEPTION\_RECORD (которое доступно через GetExceptionInformation()).
  + Это позволяет передать контекстную информацию об ошибке (например, указатели, идентификаторы, числовые значения) вместе с самим исключением.
  + Если nNumberOfArguments равен 0, то pArguments может быть NULL.

Сценарии использования RaiseException:

1. **Сигнализация об ошибках между слоями/модулями:**
   * Низкоуровневая функция обнаруживает ошибку, которую не может обработать сама. Она бросает исключение.
   * Высокоуровневый код (например, в UI-слое или в основной логике приложения) ловит это исключение и решает, что делать: показать сообщение пользователю, записать в лог, попытаться выполнить альтернативное действие.
2. **Преобразование кодов ошибок в исключения:**
   * Если вы используете библиотеку, которая возвращает коды ошибок, вы можете написать "обертку", которая проверяет код ошибки и, если это ошибка, вызывает RaiseException с более информативным кодом исключения и параметрами.
3. **Логирование и мониторинг:**
   * Можно настроить обработчик верхнего уровня, который ловит все (или определенные) программные исключения и записывает их в системный журнал Windows (Event Log) или в файл лога приложения. Это помогает в пост-анализе сбоев.
4. **Уведомление о фатальных ошибках:**
   * Если приложение входит в состояние, из которого невозможно корректно восстановиться (например, повреждение критических данных), оно может сгенерировать фатальное исключение. Обработчик верхнего уровня может попытаться сохранить какие-либо данные пользователя (если возможно) и корректно завершить приложение, вместо того чтобы просто "упасть" или продолжать работать некорректно.
5. **Реализация C++ исключений:** Как мы узнаем позже, компилятор MSVC использует RaiseException для реализации C++ throw.

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:**



**UnhandledExceptionFilter:**

* Это последняя линия обороны *внутри вашего процесса*, предоставляемая системой.
* Перед тем как окончательно "убить" ваше приложение (или поток), ОС вызывает эту функцию.
* **pExceptionInfo**: Передает ту же информацию об исключении (EXCEPTION\_POINTERS), что и GetExceptionInformation(). **Стандартное поведение UnhandledExceptionFilter:**
* **Проверяет отладчик:**
  1. Если процесс отлаживается (или настроен JIT-отладчик), предлагает пользователю начать отладку. Если пользователь согласен, управление передается отладчику.
* **Показывает диалог ошибки:**
  1. Если отладчик не запущен/не выбран, отображает стандартное окно Windows "Ошибка приложения".
* **Завершает процесс:**
  1. После закрытия диалога (или если он подавлен) принудительно завершает "сбойнувший" процесс.

**SetUnhandledExceptionFilter** позволяет вам зарегистрировать **свою собственную функцию**, которая будет вызываться вместо (или до, в зависимости от реализации) стандартного UnhandledExceptionFilter всякий раз, когда в любом потоке вашего процесса происходит необработанное исключение.



**pTopLevelExceptionFilter**: Это указатель на вашу функцию.

**Ваша функция-фильтр (MyCustomUnhandledFilter):**

* Должна иметь тот же прототип, что и UnhandledExceptionFilter.
* Получает PEXCEPTION\_POINTERS с информацией об исключении.
* **Что она может делать?**
  + **Запись в лог/создание дампа:** Собрать максимум информации об исключении (код, адрес, регистры из CONTEXT, стек вызовов) и записать это в файл или отправить на сервер.
  + **Показать свое окно ошибки:** Вместо стандартного системного.
  + **Попытаться сохранить данные пользователя (очень осторожно!).**
  + **Решить, что делать дальше:**
    - Вернуть EXCEPTION\_EXECUTE\_HANDLER: Если вы хотите, чтобы процесс завершился после вашей обработки (например, после создания дампа). Обычно после этого вызывают ExitProcess().
    - Вернуть EXCEPTION\_CONTINUE\_SEARCH: Если вы хотите, чтобы после вашей функции отработал стандартный UnhandledExceptionFilter (например, чтобы отладчик получил уведомление, если он подключен).
    - Вернуть EXCEPTION\_CONTINUE\_EXECUTION (КРАЙНЕ НЕ РЕКОМЕНДУЕТСЯ на этом уровне, так как процесс, скорее всего, нестабилен).
* **Возвращаемое значение SetUnhandledExceptionFilter:** Возвращает указатель на предыдущий зарегистрированный фильтр. Это полезно, если вы хотите временно установить свой фильтр, а потом восстановить старый.
* **Важное замечание:** Есть некоторые типы исключений (например, связанные с повреждением стека), которые могут быть настолько серьезными, что ваш пользовательский UnhandledExceptionFilter может даже не быть вызван, или его выполнение будет ограничено. Также, после некоторых обновлений безопасности Windows, поведение SetUnhandledExceptionFilter стало более строгим, чтобы предотвратить злоупотребления (например, когда вредоносное ПО пытается скрыть свои сбои).

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:



# Что лучше использовать: SEH или исключения С++?

**SEH – механизм операционной системы, доступный в любом языке программирования (в Windows, с поддержкой компилятора или WinAPI), а исключения С++ поддерживаются только в C++**

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

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

**Например, когда Вы создаете С++-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 С++-оператор throw, компилятор генерирует вызов Windows-функции RaiseException, и значение переменной, указанной в throw, передается этой функции как дополнительный аргумент**

Отлично. Я подготовлю подробное сравнение Structured Exception Handling (SEH) и механизма исключений C++ с фокусом на всех аспектах: типы обрабатываемых исключений, производительность, переносимость, безопасность, взаимодействие с внешним кодом, интеграцию с ОС и компилятором, ограничения и практические рекомендации по применению. Я сообщу, когда материал будет готов.

**Сравнение SEH и C++-исключений**

Ниже приведена таблица основных отличий между Structured Exception Handling (SEH) и стандартными исключениями C++:

| **Аспект** | **SEH** | **C++-исключения** |
| --- | --- | --- |
| **Обрабатываемые исключения**(типы) | Аппаратные ошибки (деление на ноль, нарушение доступа и др.), системные исключения Windows и программные исключения (генерируемые ОС/приложениями). Все они передаются единственным кодом (GetExceptionCode). Возможен и пользовательский вызов через RaiseException. | Только те события, которые явно брошены оператором throw – любые объекты, обычно наследующие std::exception. Такие исключения имеют свой тип и при разворачивании гарантированно вызывают деструкторы локальных объектов (RAII). |
| **Типизация и идентификация** | Не типизированы: все SEH-исключения описываются одним числовым кодом (например, EXCEPTION\_ACCESS\_VIOLATION). Обработчик выбирается по фильтру; по умолчанию C++-деструкторы не вызываются (работают только с /EHa). | Типизированы: выбор catch происходит по типу брошенного объекта. Автоматические объекты уничтожаются («разворачивается» стек) при любом C++-исключении. Нет встроенных кодов ошибок — вместо этого используется механизм поиска подходящего обработчика по иерархии типов. |
| **Синтаксис** | Специфичен для Windows/MSVC: конструкции \_\_try / \_\_except / \_\_finally. Не является частью стандарта C/C++. | Стандартный для C++: конструкции try { … } catch (…) { … }. Отсутствует finally (надежный освобождение ресурсов обеспечивается RAII). |
| **Переносимость** | Только Windows (MSVC, а также частично Clang/Intel на Windows). На других ОС SEH недоступен. | Стандарт C++ — поддерживается всеми компиляторами и ОС. Применение C++-исключений обеспечивает переносимость кода. |
| **Безопасность и надёжность** | Позволяет перехватывать критические сбои (напр., AV), но продолжать работу после них рискованно: состояние программы может быть повреждено. Кроме того, по умолчанию в MSVC при разворачивании SEH не вызываются деструкторы C++-объектов (исправляется флагом /EHa). | Исключения C++ безопасны в том смысле, что высвобождение ресурсов обеспечивается RAII. Ошибки обрабатываются явно в коде; необработанные исключения вызывают std::terminate. Аппаратные сбои C++-поток обычно вообще не ловит. |
| **Совместимость с C/C++ и наследием** | Любой код (C/asm) может по неосторожности вызвать SEH-исключение (напр. обнуленный указатель), и его можно перехватить через \_\_try/\_\_except. Windows даже позволяет устанавливать глобальные обработчики (UnhandledExceptionFilter). MSDN отмечает: «Если C-код использует SEH, его можно смешивать с C++-обработкой исключений». | Код на C не видит C++-исключений и не может их ловить (требуется обвязка). Для межъязыкового взаимодействия с C++-исключениями используют std::exception\_ptr, std::current\_exception или старые setjmp/longjmp. |

**1. Типы исключений**

* **SEH** — это механизм Windows для обработки аппаратных и программных исключений. Он ловит события, генерируемые процессором (деление на ноль, доступ к защищенной памяти и т.п.) или операционной системой. Все SEH-исключения представляются единым кодом (например, EXCEPTION\_ACCESS\_VIOLATION). Кроме того, приложение может самостоятельно вызвать SEH-исключение через функцию RaiseException, задав свой код.
* **C++-исключения** возникают только тогда, когда программист явно использует throw. Можно бросать объекты любых типов (желательно производные от std::exception). C++-исключения не имеют общих кодов — каждый бросаемый объект определяется своим типом, и обработка происходит по совпадению типов. При этом при разворачивании стека гарантированно вызываются деструкторы локальных объектов. Аппаратные сбои процессор не превращает автоматически в C++-исключения (если не настроен \_set\_se\_translator или не используется /EHa), поэтому нарушения памяти или т.п. обычно приводят к аварийному завершению.

setjmp и longjmp — это пара функций из стандартной библиотеки C (заголовочный файл <setjmp.h>), которые предоставляют механизм для выполнения **нелокальных переходов** (non-local gotos). Это способ передать управление из одной точки программы в другую, которая может находиться в другой функции в стеке вызовов, минуя обычный механизм возврата из функций.

Представьте это как создание "точки сохранения" в вашей программе (setjmp) и возможность "телепортироваться" обратно к этой точке из другого места (longjmp), даже из глубоко вложенной функции.

**1. int setjmp(jmp\_buf env)**

* **jmp\_buf env**: Это специальный тип данных (обычно массив), который setjmp использует для сохранения текущего "контекста выполнения". Контекст включает информацию о состоянии процессора, такую как указатель стека, значения регистров и т.д. Вы просто объявляете переменную этого типа и передаете ее в setjmp.
* **Действие**:
  1. Когда setjmp вызывается **впервые (напрямую)**, она сохраняет текущий контекст выполнения в env.
  2. В этом случае setjmp возвращает 0.
* **Возвращаемое значение (ключевой момент!)**:
  1. При прямом вызове: 0.
  2. Когда управление возвращается к этому setjmp через вызов longjmp(env, value): setjmp возвращает значение value, переданное в longjmp. **Однако, если value было 0, setjmp все равно вернет 1** (чтобы не путать с первоначальным возвратом 0).

**2. void longjmp(jmp\_buf env, int value)**

* **jmp\_buf env**: Та же самая переменная jmp\_buf, которая была использована в соответствующем вызове setjmp.
* **int value**: Целочисленное значение, которое "вернется" из setjmp после перехода. Если передать 0, setjmp вернет 1.
* **Действие**:
  1. longjmp восстанавливает контекст выполнения, сохраненный в env предыдущим вызовом setjmp.
  2. Выполнение программы немедленно переходит к точке сразу после вызова setjmp, как если бы setjmp только что вернул значение value (или 1, если value было 0).
  3. longjmp **никогда не возвращает управление в точку своего вызова**. Она "разматывает" стек вызовов до уровня функции, где был сделан setjmp.

Стек вызовов:

| function\_C | <- Вершина стека, текущая выполняемая функция

| function\_B |

| function\_A | (здесь был setjmp)

| main |

+------------+

Затем, в function\_C(), происходит вызов longjmp(jump\_buffer, 1). Что делает longjmp:

1. **Немедленно прекращает выполнение function\_C().** Код в function\_C() после longjmp не выполняется.
2. **"Снимает" тарелку C со стека.** Локальные переменные function\_C() уничтожаются (для C, для C++ без вызова деструкторов).
3. **Немедленно прекращает выполнение function\_B().** Код в function\_B() после того места, откуда была вызвана function\_C(), не выполняется.
4. **"Снимает" тарелку B со стека.** Локальные переменные function\_B() уничтожаются.
5. **Доходит до function\_A() (где был setjmp).**
6. **Восстанавливает состояние процессора (регистры, указатель стека) так, как оно было на момент вызова setjmp в function\_A().**
7. **Выполнение продолжается в function\_A() сразу после setjmp, как если бы setjmp только что вернул значение, переданное в longjmp (в нашем случае, 1).**

Блок \_\_finally **гарантированно** выполняется всякий раз, когда поток управления покидает связанный с ним блок \_\_try, независимо от того, как он его покидает:

1. **Нормальное завершение:** Код в \_\_try выполнился до конца.
2. **Исключение SEH:** Внутри \_\_try (или в функции, вызванной из \_\_try) произошло исключение, и система ищет обработчик (\_\_except). Перед передачей управления в фильтр или обработчик (или если исключение "пролетает мимо"), все \_\_finally на пути раскрутки должны выполниться.
3. **return из \_\_try:** Перед фактическим возвратом из функции выполнится \_\_finally.
4. **goto из \_\_try (или \_\_except) наружу:** Перед переходом к метке выполнится \_\_finally, если goto покидает область \_\_try.
5. **break или continue из \_\_try (внутри цикла):** Перед выходом из цикла или переходом к следующей итерации выполнится \_\_finally, если break/continue покидают область \_\_try.
6. **longjmp:** Если longjmp "перепрыгивает" через фреймы стека, которые содержат активные \_\_try/\_\_finally блоки, эти \_\_finally блоки должны выполниться в процессе раскрутки.

* **ExceptionRecord->ExceptionFlags & EXCEPTION\_NESTED\_CALL**: Это побитовая операция "И". EXCEPTION\_NESTED\_CALL — это флаг, который устанавливается системой, если исключение произошло во время обработки другого исключения (например, внутри блока \_\_except или \_\_finally, или внутри другого фильтра).
* **((ExceptionRecord->ExceptionFlags & EXCEPTION\_NESTED\_CALL) == 0)**: Проверяет, что флаг EXCEPTION\_NESTED\_CALL **не установлен**. То есть, это "первичное" исключение, а не вложенное.

**VirtualAlloc**:

* + Это мощная функция WinAPI для управления виртуальной памятью процесса. Она может:
    - Резервировать диапазон виртуальных адресов (MEM\_RESERVE).
    - Передавать (commit) физическую память для ранее зарезервированного диапазона (MEM\_COMMIT).

**Когда бы E1 было "подавлено" E2?**

E1 было бы фактически подавлено (или, точнее, E2 стало бы доминирующим необработанным исключением), если бы:

* Исключение E2, возникшее в \_\_finally, **не было бы обработано** никаким \_\_except (ни вложенным, ни на более высоком уровне, который мог бы перехватить E2). В этом случае E2 распространилось бы вверх, и процесс раскрутки стека для E1 был бы прерван уже из-за необработанного E2. Система бы уже разбиралась с E2 как с фатальной ошибкой.
* Если бы внутри \_\_finally (например, после возникновения E2, или даже вместо E2) был бы выполнен longjmp. longjmp из \_\_finally – это специальный случай, который прерывает нормальную обработку исключений.