**Что такое стек вызовов?** Когда твоя программа работает, она использует область памяти, называемую **стеком**.

**Вызов функции:**

Когда функция A вызывает функцию B:

Адрес возврата (куда вернуться в A после завершения B) помещается в стек.

Аргументы для B могут быть переданы через стек (или регистры, или смешанно).

В стеке выделяется место для локальных переменных функции B.

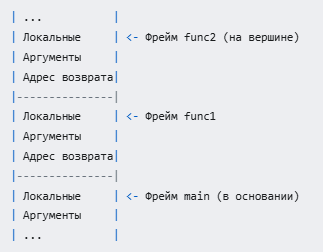
Этот набор данных в стеке (адрес возврата, сохраненные регистры, место для аргументов и локальных переменных) для одного активного вызова функции называется **стековым фреймом** (stack frame) или **кадром активации** (activation record).

**Структура стека вызовов:**

main() вызывает func1(). Стековый фрейм для func1 кладется "поверх" фрейма main.

func1() вызывает func2(). Стековый фрейм для func2 кладется "поверх" фрейма func1.

Сейчас на вершине стека — фрейм func2. Под ним — func1. Еще ниже — main.



**Завершение функции:** Когда функция func2 завершается, ее стековый фрейм "снимается" со стека (указатель стека возвращается на прежнее место). Управление возвращается в func1 по сохраненному адресу возврата.

**SEH и стековые фреймы:** Если функция содержит \_\_try/\_\_except или \_\_try/\_\_finally, компилятор добавляет специальную информацию к ее стековому фрейму (или в отдельные таблицы, связанные с адресами кода этой функции). Эта информация говорит системе, что эта функция имеет дело с SEH, и указывает, где находятся фильтр \_\_except или код \_\_finally.

**Пример стековых фреймов:**

void funcC() {

int c\_var = 30;

// ... точка возникновения исключения ...

// Предположим, здесь возникает исключение

\*((int\*)0) = 123; // Access Violation

}

void funcB() {

int b\_var = 20;

\_\_try {

funcC();

}

\_\_finally {

printf("Finally в funcB\n"); // Код finally для funcB

}

}

void funcA() {

int a\_var = 10;

\_\_try {

funcB();

}

\_\_except (MyFilter(GetExceptionInformation())) { // Фильтр и обработчик для funcA

printf("Except в funcA\n");

}

}

int main() {

funcA();

return 0;

}

Когда исключение происходит в funcC:  
Стек вызовов (упрощенно, вершина сверху):

1. Фрейм funcC (содержит c\_var)
2. Фрейм funcB (содержит b\_var и информацию о \_\_finally)
3. Фрейм funcA (содержит a\_var и информацию о \_\_except(MyFilter))
4. Фрейм main

**Поиск Нужного Обработчика (\_\_except)**

Когда в funcC происходит Access Violation:

1. **Приостановка выполнения:** Выполнение funcC немедленно останавливается.
2. **Начало диспетчеризации:** ОС (диспетчер исключений) берет управление.
3. **Поиск снизу вверх по стеку:** Диспетчер начинает просматривать стековые фреймы, двигаясь от текущего фрейма (funcC) вверх (к funcB, затем к funcA, затем к main и т.д.).
4. **Проверка фрейма funcC:**
   * Диспетчер смотрит на информацию, связанную со стековым фреймом funcC.
   * В funcC нет \_\_try/\_\_except. Значит, funcC не может обработать это исключение.
   * Поиск продолжается.
5. **Проверка фрейма funcB:**
   * Диспетчер переходит к фрейму funcB.
   * Он видит, что funcB имеет \_\_try/\_\_finally. Блоки \_\_finally не *обрабатывают* исключения в смысле их "поглощения" (как \_\_except), они лишь выполняют код очистки. Поэтому сам по себе \_\_finally не останавливает поиск обработчика \_\_except.
   * Поиск продолжается.
6. **Проверка фрейма funcA:**
   * Диспетчер переходит к фрейму funcA.
   * Он видит, что funcA имеет конструкцию \_\_try/\_\_except(MyFilter(...)). Это кандидат!
   * **Вызов фильтра:** Диспетчер вызывает выражение-фильтр, то есть MyFilter(GetExceptionInformation()).
     + GetExceptionInformation() предоставляет фильтру MyFilter информацию о текущем исключении (Access Violation из funcC).
     + Допустим, наш MyFilter выглядит так:

int MyFilter(LPEXCEPTION\_POINTERS pExceptionInfo) {

if (pExceptionInfo->ExceptionRecord->ExceptionCode == STATUS\_ACCESS\_VIOLATION) {

printf("Фильтр в funcA: поймал Access Violation! Будем обрабатывать.\n");

return EXCEPTION\_EXECUTE\_HANDLER; // Да, этот \_\_except обработает исключение

}

printf("Фильтр в funcA: это не Access Violation, ищем дальше.\n");

return EXCEPTION\_CONTINUE\_SEARCH; // Нет, ищем другой обработчик

}

* + - В нашем случае код исключения – STATUS\_ACCESS\_VIOLATION. Фильтр MyFilter вернет EXCEPTION\_EXECUTE\_HANDLER.

1. **Обработчик найден!**
   * Поскольку фильтр MyFilter вернул EXCEPTION\_EXECUTE\_HANDLER, диспетчер исключений определил, что \_\_except блок в funcA будет обрабатывать это исключение.
   * **Поиск обработчика на этом завершается.**

**Что если бы MyFilter вернул EXCEPTION\_CONTINUE\_SEARCH?**  
Диспетчер проигнорировал бы \_\_except в funcA и продолжил бы поиск вверх по стеку (к main). Если бы и в main не нашлось обработчика, и не было бы отладчика или обработчика верхнего уровня, программа бы аварийно завершилась.

**Раскрутка Стека для \_\_finally (и для выполнения \_\_except)**

Итак, диспетчер нашел обработчик в funcA (фильтр MyFilter вернул EXCEPTION\_EXECUTE\_HANDLER). **Но перед тем, как выполнить код внутри \_\_except блока funcA, должна произойти раскрутка стека.**

**Что такое раскрутка стека (Stack Unwinding)?**

**Это процесс корректного "сворачивания" стековых фреймов, которые находятся между местом возникновения исключения и найденным обработчиком**. "Корректного" означает, что все необходимые блоки \_\_finally должны быть выполнены.

**Процесс раскрутки в нашем примере:**

1. **Место исключения:** funcC.
2. **Найденный обработчик:** \_\_except в funcA.
3. **Стековые фреймы между ними (включительно до фрейма перед обработчиком):** funcC и funcB.
4. **Шаг 1 раскрутки: фрейм funcC**
   * Диспетчер смотрит на фрейм funcC. Есть ли у него \_\_finally? Нет.
   * Фрейм funcC считается "раскрученным". Локальные переменные funcC (если они были объектами C++ с деструкторами, и компилятор это поддерживает для SEH) были бы уничтожены.
5. **Шаг 2 раскрутки: фрейм funcB**
   * Диспетчер смотрит на фрейм funcB. Есть ли у него \_\_finally? **Да!**
   * **Выполнение \_\_finally в funcB:** Код внутри этого \_\_finally блока выполняется. На консоль будет выведено: Finally в funcB.
   * После выполнения \_\_finally, фрейм funcB считается "раскрученным".
6. **Раскрутка завершена до funcA:** Все фреймы "ниже" funcA (т.е. funcC и funcB) теперь раскручены.
7. **Выполнение \_\_except блока:**
   * Теперь, когда стек корректно раскручен до фрейма funcA, управление передается коду внутри \_\_except блока funcA
   * На консоль будет выведено: Except в funcA.
8. **Продолжение выполнения:**
   * После завершения \_\_except блока, выполнение программы продолжается с инструкции, следующей за всей конструкцией \_\_try/\_\_except в funcA. Затем funcA завершится, и управление вернется в main.

Проект windows\_seh\_tests-main представляет собой набор тестов для валидации реализации SEH в Windows. SEH — это механизм языка C/C++, позволяющий обрабатывать аппаратные и программные исключения с помощью ключевых слов \_\_try, \_\_except, \_\_finally и \_\_leave.

# README

SEH работает благодаря взаимодействию трех частей:

1. **Среда выполнения ОС Windows (Windows OS Runtime):**
   * **Роль:** Предоставляет базовые услуги – как найти обработчик ошибки и как безопасно "размотать" стек вызовов.
2. **Компилятор (Compiler):**
   * **Роль 1:** Генерирует **метаданные** для ОС. Это информация о каждой функции: где она хранит важные данные на стеке, сколько места занимает. Это нужно ОС для правильной раскрутки стека. (На x64/ARM это таблицы, на x86 раньше было иначе).
   * **Роль 2:** Генерирует специальный код (**фанклеты** – funclets). Это маленькие кусочки кода, которые соответствуют вашим блокам \_\_except (фильтр-выражению) и \_\_finally. C-Runtime вызывает эти фанклеты.
3. **Среда выполнения C (C-Runtime):**
   * **Роль:** Связующее звено между ОС и кодом, который сгенерировал компилятор. Вызывает фанклеты. Управляет потоком SEH внутри одной функции (например, что делать при \_\_leave).
   * **Важно:** C-Runtime также поддерживает C++ исключения (throw/catch). Но C++ catch по умолчанию **не ловит** ошибки ОС/процессора (вроде Access Violation). SEH (\_\_except) – ловит.

Описание Тестовых Наборов

* **XCPT4**
  + Фокус: Проверка компилятора и C-Runtime.
  + Что тестирует: В основном, SEH *внутри одной функции* (локальные исключения). Немного затрагивает и межфункциональные.
  + Результат: "passed" (пройдено) и "skipped" (пропущено) – это нормально. "failed" (провалено) или креш – плохо. Пропуски бывают, т.к. не все тесты актуальны для всех архитектур CPU.
* **NESTED\_COLLIDED**
  + Фокус: Два сложных случая:
    1. Вложенные исключения (Nested Exception): Ошибка возникает *внутри фильтра \_\_except*, который сам обрабатывал предыдущую ошибку.
    2. Конфликтующие раскрутки (Collided Unwind): Ошибка возникает *внутри блока \_\_finally*, который выполнялся во время раскрутки стека от предыдущей ошибки.
  + Результат: Два сообщения "PASSED."
* **XFRAME\_TEST**
  + Фокус: Обработка исключений *между разными модулями* (EXE и DLL).
  + Особенно важно: Проверка совместимости, если EXE и DLL скомпилированы *разными компиляторами* (или разными версиями одного).
  + Как использовать для теста своего компилятора:
    1. Скомпилировать EXE стабильным компилятором MSVC, а DLL – вашим компилятором. Запустить.
    2. Скомпилировать DLL стабильным компилятором MSVC, а EXE – вашим компилятором. Запустить.
  + Результат: Два сообщения "PASSED." (одно для "обработанных" исключений, другое для "возобновленных").

# nested\_collided/nestcol.c

volatile BOOLEAN BreakOnEachStep = FALSE;

BOOLEAN: TRUE = 1, FALSE = 0.

По умолчанию FALSE, то есть НЕТ прерывания на каждом шаге.

volatile: говорит компилятору, что значение этой переменной может измениться в любой момент времени извне текущего потока кода. Поэтому компилятор не должен делать предположений о значении этой переменной и не должен оптимизировать доступ к ней. Это позволяет разработчику, отлаживающему тесты, вручную изменить значение BreakOnEachStep на TRUE (например, в отладчике) во время выполнения программы.

ULONG64 TestAccumulator;

Эта переменная используется для **отслеживания последовательности выполнения шагов теста**. Каждый "шаг" в тесте (обозначенный макросом **TEST\_STEP**) будет умножать текущее значение TestAccumulator на уникальное простое число. В конце теста итоговое значение TestAccumulator сравнивается с ожидаемым произведением. Если они совпадают, это с высокой вероятностью означает, что все шаги были выполнены в правильном порядке и правильное количество раз. Использование простых чисел помогает гарантировать, что разный порядок или пропуск/повтор шагов приведет к разным итоговым значениям.

Дальше три макроса:

**Макрос START\_TEST** вызывается в начале каждого логического теста для сброса состояния отслеживания и установки начальной точки для аккумулятора.

#define START\_TEST \

{ \

if (BreakOnEachStep) { \

\_\_debugbreak(); \

} \

\

TestAccumulator = 1; \

}

Проверяется значение глобальной переменной BreakOnEachStep.

* Если BreakOnEachStep равно TRUE, вызывается функция \_\_debugbreak().
* **\_\_debugbreak():** Это специальная функция (интринсик компилятора MSVC), которая генерирует точку останова (breakpoint) в коде. Если программа запущена под отладчиком, выполнение остановится на этой строке. Если отладчика нет, это обычно приводит к аварийному завершению программы (исключение STATUS\_BREAKPOINT).

Присваивает глобальной переменной TestAccumulator начальное значение 1. Это важно, так как последующие шаги будут умножать это значение

Макрос **TEST\_STEP(N)** вызывается в различных точках тестового кода, чтобы отметить, что определенный этап был достигнут. N должно быть уникальным простым числом для этого шага.

#define TEST\_STEP(\_\_\_PRIME) \

{ \

if (BreakOnEachStep) { \

\_\_debugbreak(); \

} \

\

TestAccumulator \*= (\_\_\_PRIME); \

}

Макрос принимает один аргумент \_\_\_PRIME. Ожидается, что это будет простое число.

Та же проверка, что и в START\_TEST. Позволяет остановить выполнение на конкретном шаге, если отладка включена.

Текущее значение TestAccumulator умножается на переданное простое число \_\_\_PRIME. Скобки вокруг \_\_\_PRIME важны для макросов, чтобы избежать проблем с порядком операций, если бы \_\_\_PRIME было сложным выражением (хотя здесь это обычно просто число).

Макрос **END\_TEST**(expected\_value) вызывается в конце каждого логического теста для проверки, что TestAccumulator содержит правильное контрольное значение.

#define END\_TEST(\_\_\_ANSWER) \

{ \

if (BreakOnEachStep) { \

\_\_debugbreak(); \

} \

\

if (TestAccumulator != (\_\_\_ANSWER)) { \

\_\_debugbreak(); \

} \

}

Один параметр. Снова та же проверка для отладки. Сравнивает текущее значение TestAccumulator с ожидаемым значением \_\_\_ANSWER. Если значения не совпадают, вызывается \_\_debugbreak(). Это приведет к остановке под отладчиком (показывая место сбоя) или к крешу.

void

just\_raise (void)

{

TEST\_STEP(3)

RaiseException(0x4000, 0, 0, NULL);

}

Вызывается макрос TEST\_STEP с аргументом 3.

**RaiseException(0x4000, 0, 0, NULL);**: Это вызов функции Windows API RaiseException. Эта функция позволяет программе самой сгенерировать (возбудить) программное исключение.

* dwExceptionCode (DWORD): Код исключения. Здесь это 0x4000. Это пользовательский код исключения.
* dwExceptionFlags (DWORD): Флаги исключения. Здесь 0.
  + Если 0, то исключение считается "продолжаемым" (continuable). Это означает, что обработчик исключений может попытаться исправить проблему и вернуть EXCEPTION\_CONTINUE\_EXECUTION, чтобы программа возобновила выполнение с того места, где произошло исключение.
* nNumberOfArguments (DWORD): Количество аргументов в следующем массиве. Здесь 0.
* lpArguments (const ULONG\_PTR \*): Указатель на массив дополнительных аргументов для исключения. Здесь NULL, так как nNumberOfArguments равен 0.

**Что происходит после вызова RaiseException?**

* Выполнение текущей функции just\_raise немедленно прекращается.
* Операционная система берет управление на себя.
* ОС начинает процесс диспетчеризации исключений: она ищет подходящий обработчик для исключения с кодом 0x4000. Поиск начинается с текущего стекового кадра (функции just\_raise, затем функции, которая ее вызвала, и так далее вверх по стеку вызовов).
* ОС ищет блоки \_\_try/\_\_except (или другие механизмы обработки исключений, например, отладчик).

int

execute\_handler\_filter (EXCEPTION\_POINTERS \*pExcept)

{

UNREFERENCED\_PARAMETER(pExcept);

TEST\_STEP(5)

return EXCEPTION\_EXECUTE\_HANDLER;

}

int: Как **и любая фильтр-функция в SEH**, она должна возвращать целочисленное значение типа int. Это значение определяет, как система должна поступить с исключением.

**EXCEPTION\_POINTERS** \*pExcept: Это стандартный параметр для фильтр-функций SEH. В коде \_\_except выражение фильтра обычно выглядит так: \_\_except (имя\_фильтра(GetExceptionInformation())). Функция GetExceptionInformation() (которую можно вызывать **только** внутри выражения фильтра SEH) возвращает указатель типа LPEXCEPTION\_POINTERS (что эквивалентно EXCEPTION\_POINTERS\*), который затем передается в фильтр-функцию.

**UNREFERENCED\_PARAMETER**: Это макрос, обычно определенный в windows.h (или winnt.h). Он предназначен для подавления предупреждений компилятора о неиспользуемых параметрах.

**return EXCEPTION\_EXECUTE\_HANDLER;**: фильтр-функция сообщает операционной системе (и C-Runtime), что ассоциированный с этим фильтром блок \_\_except { ... } **должен быть выполнен**.

int

nested\_exception\_filter (EXCEPTION\_POINTERS \*pExcept)

{

TEST\_STEP(7)

if (pExcept->ExceptionRecord->ExceptionCode == 0x4000) {

RaiseException(0x4001, 0, 0, NULL);

}

return EXCEPTION\_CONTINUE\_SEARCH;

}

Эта функция является фильтром для SEH-конструкции \_\_try/\_\_except. Её главная задача в этом тесте — создать **вложенное исключение**.

Фильтр проверяет, для какого исключения он был вызван. Если код исключения — 0x4000 (это "первичное" исключение, которое мы ожидаем от just\_raise), то выполняется тело if.

**RaiseException(0x4001, 0, 0, NULL);**:

* Это сердцевина механизма "вложенного исключения". Прямо во время работы фильтра, который анализирует исключение 0x4000, программа генерирует совершенно новое исключение 0x4001.
* **Поток управления при RaiseException**: Как только RaiseException вызывается, текущий процесс обработки исключения 0x4000 приостанавливается. Система немедленно переключается на обработку нового, "вложенного" исключения 0x4001. Начинается новый цикл поиска обработчика, но уже для 0x4001.

**Сценарий 1: Фильтр вызван для другого исключения (например, для 0x4001 при повторном входе).**

1. TEST\_STEP(7) выполняется.

2. if не срабатывает (т.к. код не 0x4000).

3. Фильтр немедленно возвращает EXCEPTION\_CONTINUE\_SEARCH. Это означает, что \_\_except блок, к которому относится этот фильтр, не будет обрабатывать 0x4001. Исключение 0x4001 продолжит распространяться вверх по стеку.

**Сценарий 2: Фильтр вызван для 0x4000.**

**Тут дальше будут фигурировать другие функции, которые ещё не обсуждались**

1. **TEST\_STEP(7) выполняется.**
2. **if (pExcept->ExceptionRecord->ExceptionCode == 0x4000) условие истинно.**
3. **RaiseException(0x4001, 0, 0, NULL); выполняется.**
   * В этот момент происходит **Nested Dispatch** (Вложенная Диспетчеризация).
   * Система Windows **приостанавливает** любую дальнейшую обработку исключения 0x4000. Она не забывает о нем полностью, но откладывает его в сторону.
   * **Главным приоритетом становится обработка нового исключения 0x4001.** Система немедленно начинает новый поиск обработчика (новый "проход диспетчера") для 0x4001.
4. **Поиск обработчика для 0x4001:**
   * Система смотрит вверх по стеку вызовов. Первый же SEH-фрейм, который она встречает, — это тот самый \_\_except (nested\_exception\_filter(...)) в функции nested\_exception.
   * Поэтому nested\_exception\_filter вызывается **второй раз**, но теперь уже с pExcept->ExceptionRecord->ExceptionCode == 0x4001.
     + TEST\_STEP(7) выполняется (второй раз для этого фильтра).
     + Условие if (... == 0x4000) теперь **ложно**.
     + Новое исключение не генерируется.
     + nested\_exception\_filter возвращает EXCEPTION\_CONTINUE\_SEARCH **для исключения 0x4001**.
   * Поскольку фильтр для 0x4001 вернул EXCEPTION\_CONTINUE\_SEARCH, \_\_except блок в nested\_exception не обрабатывает 0x4001.
   * Исключение 0x4001 "вылетает" из функции nested\_exception и попадает в main.
5. **Обработка 0x4001 в main:**
   * В main есть \_\_try { ... } \_\_except (execute\_handler\_filter(...)).
   * execute\_handler\_filter вызывается для 0x4001.
     + TEST\_STEP(5) выполняется.
     + execute\_handler\_filter возвращает EXCEPTION\_EXECUTE\_HANDLER.
   * Система решает, что \_\_except блок в main будет обрабатывать 0x4001.
   * **Раскрутка стека для 0x4001:** Прежде чем выполнить тело \_\_except в main, происходит раскрутка стека до этого уровня. Это означает, что все \_\_finally блоки между местом возникновения 0x4001 (это фильтр nested\_exception\_filter) и main должны быть выполнены. В нашем случае таких \_\_finally нет на этом пути.
   * Тело \_\_except в main выполняется: TEST\_STEP(19).
   * **Обработка 0x4001 на этом полностью завершена.** Исключение считается обработанным.
6. **Что происходит с первоначальным исключением 0x4000?**
   * Когда вложенное исключение (0x4001), возникшее в фильтре первого исключения (0x4000), **успешно обрабатывается** (т.е. какой-то \_\_except блок выполняется для него, как в нашем случае \_\_except в main для 0x4001), то **первоначальное исключение (0x4000) обычно отбрасывается (dismissed).**
   * Система не "возвращается к вопросу, что делать с 0x4000" после того, как 0x4001 было полностью обработано через EXCEPTION\_EXECUTE\_HANDLER.
   * Фактически, обработка 0x4001 заменяет собой дальнейшую обработку 0x4000.
   * Значение return EXCEPTION\_CONTINUE\_SEARCH; из *первого* вызова nested\_exception\_filter (которое формально относилось к 0x4000) становится нерелевантным, потому что поток обработки был "перехвачен" и перенаправлен на 0x4001.

void

nested\_exception (void)

{

\_\_try

{

just\_raise();

}

\_\_except (nested\_exception\_filter(GetExceptionInformation()))

{

TEST\_STEP(11)

}

}

Выполняется just\_raise(). Генерируется исключение с кодом 0x4000.

Поскольку в \_\_try произошло исключение 0x4000, система обращается к \_\_except блоку. Выполняется выражение-фильтр: nested\_exception\_filter(GetExceptionInformation()).

**Первый вызов nested\_exception\_filter (для исключения 0x4000):**

Условие if (pExcept->ExceptionRecord->ExceptionCode == 0x4000) **истинно**.

RaiseException(0x4001, ...): Генерируется **вложенное исключение 0x4001**.

Система немедленно начинает искать обработчик для 0x4001.

**Первый кандидат** на пути поиска – это тот же самый \_\_except (nested\_exception\_filter(...)) блок в функции nested\_exception! Да, SEH-структура может попытаться обработать исключение, возникшее в её собственном фильтре.

**Второй вызов nested\_exception\_filter (теперь для исключения 0x4001):**

Условие if (pExcept->ExceptionRecord->ExceptionCode == 0x4000) **ложно** (т.к. текущий ExceptionCode это 0x4001).

nested\_exception\_filter возвращает EXCEPTION\_CONTINUE\_SEARCH (это значение относится к судьбе 0x4001).

**Тело \_\_except { TEST\_STEP(11) }**:

Поскольку *второй* вызов nested\_exception\_filter (для 0x4001) вернул EXCEPTION\_CONTINUE\_SEARCH, этот блок \_\_except **не выполняется**.

**Результат работы nested\_exception()**:

Функция завершается, но из нее "вылетает" необработанное исключение 0x4001. Это исключение будет передано для обработки в функцию, которая вызвала nested\_exception() (то есть, в main).

void

collided\_unwind (void)

{

\_\_try

{

just\_raise();

}

\_\_finally

{

TEST\_STEP(13)

RaiseException(0x4001, 0, 0, NULL);

}

}

Выполняется just\_raise(). Генерируется исключение с кодом 0x4000.

Поскольку в \_\_try произошло исключение 0x4000, система должна покинуть \_\_try блок.

Система (диспетчер исключений) начинает **первый проход** вверх по стеку вызовов.

Цель этого прохода — найти \_\_except блок.

На этом этапе блоки \_\_finally НЕ ВЫПОЛНЯЮТСЯ. Система только "заглядывает" в фильтры \_\_except.

Если такой \_\_except блок найден, система "запоминает" его и переходит ко второму этапу.

Если ни один фильтр не вернул EXCEPTION\_EXECUTE\_HANDLER, то исключение считается необработанным (и дальше уже вступают в дело отладчики, обработчики верхнего уровня или система завершает программу).

**Цель второго прохода — "раскрутить" стек** от места возникновения исключения до найденного обработчика, **выполняя по пути все \_\_finally блоки**, и затем выполнить тело найденного \_\_except блока.

**Выполнение \_\_finally**:

TEST\_STEP(13): (TestAccumulator станет B \* 3 \* 13).

RaiseException(0x4001, ...): **Ключевой момент для "Collided Unwind"!** Внутри блока \_\_finally, который выполняется из-за исключения 0x4000, генерируется новое исключение 0x4001.

**Эффект "столкновения" (Collision)**:

Раскрутка стека, которая происходила для исключения 0x4000, "сталкивается" с новым исключением 0x4001.

Первоначальное исключение 0x4000 больше не является главным. Система теперь фокусируется на обработке нового исключения 0x4001.

Поиск обработчика для 0x4001 начнется с текущего места (из \_\_finally блока функции collided\_unwind) и пойдет вверх по стеку.

**Результат работы collided\_unwind()**:

Функция завершается, но из нее "вылетает" необработанное исключение 0x4001. Это исключение будет передано для обработки в функцию, которая вызвала collided\_unwind() (то есть, в main).