**Общие моменты:**

* **Counter**: Переменная, используемая для отслеживания состояния выполнения теста. Её конечное значение сравнивается с ожидаемым.
* **RaiseStatus(STATUS\_...)**: Макрос, который вызывает RaiseException, генерируя SEH-исключение.
* **\*BadAddress**: Обычно NULL или некорректный указатель. Обращение к нему вызывает исключение STATUS\_ACCESS\_VIOLATION.
* **\_\_try, \_\_except, \_\_finally**: Ключевые слова SEH.
  + \_\_try: Блок кода, в котором может произойти исключение.
  + \_\_except(filter\_expression): Блок обработчика исключений. Выполняется, если в \_\_try произошло исключение, и filter\_expression вернуло EXCEPTION\_EXECUTE\_HANDLER (1).
    - EXCEPTION\_CONTINUE\_SEARCH (0): Искать другой обработчик.
    - EXCEPTION\_CONTINUE\_EXECUTION (-1): Возобновить выполнение с точки исключения.
  + \_\_finally: Блок завершения. Выполняется всегда, когда управление покидает \_\_try блок, будь то нормальное завершение, исключение, return, goto, break, continue, longjmp, или \_\_leave.
* **abnormal\_termination()**: Функция, используемая внутри \_\_finally. Возвращает TRUE, если \_\_try блок был покинут не по порядку (из-за исключения, goto, return, longjmp), и FALSE при нормальном завершении или использовании \_\_leave.
* **setjmp/longjmp**: Механизм нелокальных переходов в C. \_\_finally блоки должны корректно обрабатываться при longjmp.
* **C++ throw**: Механизм исключений C++. При использовании /EHsc (что у вас), SEH \_\_finally блоки будут выполняться во время раскрутки стека C++ исключения, но \_\_except блоки не будут перехватывать C++ исключения (для этого нужен /EHa).
* **DECLSPEC\_NOINLINE**: Гарантирует, что функция не будет встроена компилятором, что важно для тестирования вызовов функций и раскрутки стека.
* **#if defined(BAIL\_IN\_FINALLY) и т.п.**: Условная компиляция. Некоторые тесты могут быть пропущены в зависимости от этих макросов, которые обычно определяются для конкретных архитектур или конфигураций компилятора, где такое поведение разрешено или тестируется. BAIL\_IN\_FINALLY обычно означает разрешение return, goto, longjmp прямо из блока \_\_finally. NEST\_IN\_FINALLY - разрешение вложенных \_\_try в \_\_finally.

**Разбор тестов из xcpt4u.c (функция main)**

**Test 1:**

* **Цель:** Проверка простого \_\_try/\_\_finally, когда исключения нет.
* **Механизм:** Counter инкрементируется в \_\_try. \_\_finally выполняется, abnormal\_termination() вернет FALSE.
* **Ожидаемый Counter:** 2 (1 из \_\_try + 1 из \_\_finally).

**Test 2:**

* **Цель:** Проверка простого \_\_try/\_\_except, когда исключения нет.
* **Механизм:** Counter инкрементируется в \_\_try. Блок \_\_except не выполняется.
* **Ожидаемый Counter:** 1.

**Test 3:**

* **Цель:** Проверка \_\_try/\_\_except, когда исключение есть, но фильтр \_\_except (значение Counter, которое -1) заставляет продолжить выполнение (EXCEPTION\_CONTINUE\_EXECUTION).
* **Механизм:** Counter декрементируется. RaiseStatus вызывает исключение. Фильтр \_\_except(Counter) вернет -1. Выполнение возобновляется после RaiseStatus.
* **Ожидаемый Counter:** -1.

**Test 4:**

* **Цель:** Проверка \_\_try/\_\_except, когда исключение есть и фильтр \_\_except (значение Counter, которое 1) его обрабатывает (EXCEPTION\_EXECUTE\_HANDLER).
* **Механизм:** Counter инкрементируется. RaiseStatus вызывает исключение. Фильтр \_\_except(Counter) вернет 1. Блок \_\_except выполняется.
* **Ожидаемый Counter:** 2 (1 из \_\_try + 1 из \_\_except).

**Test 5:**

* **Цель:** Проверка \_\_try/\_\_except с исключением доступа к памяти (\*BadAddress), обработчик выполняется.
* **Механизм:** Counter инкрементируется. \*BlackHole += \*BadAddress; вызывает исключение. Фильтр \_\_except(Counter) вернет 1. Блок \_\_except выполняется.
* **Ожидаемый Counter:** 2.

**Test 6:**

* **Цель:** Вложенный \_\_try/\_\_finally внутри внешнего \_\_try/\_\_except. Исключение во внутреннем \_\_try.
* **Механизм:** Внутренний \_\_try: Counter = 1. RaiseStatus. Внутренний \_\_finally: abnormal\_termination() = TRUE, Counter = 2. Внешний \_\_except: Counter = 2, условие if выполняется, Counter = 3.
* **Ожидаемый Counter:** 3.

**Test 7:**

* **Цель:** Аналогично Test 6, но с исключением доступа к памяти.
* **Механизм:** Внутренний \_\_try: Counter = 1. \*BadAddress вызывает исключение. Внутренний \_\_finally: abnormal\_termination() = TRUE, Counter = 2. Внешний \_\_except: Counter = 2, условие if выполняется, Counter = 3.
* **Ожидаемый Counter:** 3.

**Test 8:**

* **Цель:** Исключение генерируется в вызываемой функции foo1.
* **Механизм:** Counter = 1. foo1 вызывает RaiseStatus(STATUS\_ACCESS\_VIOLATION). Фильтр \_\_except проверяет код исключения и обрабатывает его. Counter = 2.
* **Ожидаемый Counter:** 2.

**Test 9:**

* **Цель:** Исключение доступа к памяти в вызываемой функции foo2.
* **Механизм:** Counter = 1. foo2 вызывает \*BadAddress. Фильтр \_\_except обрабатывает. Counter = 2.
* **Ожидаемый Counter:** 2.

**Test 10:**

* **Цель:** Исключение в foo1, вызываемой из bar1, которая имеет \_\_finally.
* **Механизм:** bar1 вызывается. В bar1: foo1 вызывает RaiseStatus. \_\_finally в bar1: abnormal\_termination() = TRUE, \*Counter (переданный как аргумент) становится 99. Внешний \_\_except (в main): код исключения совпадает, Counter декрементируется до 98.
* **Ожидаемый Counter:** 98.

**Test 11:**

* **Цель:** Аналогично Test 10, но с bar2 и foo2 (доступ к \*BadAddress).
* **Механизм:** Аналогично Test 10. Counter становится 99 в \_\_finally функции bar2, затем декрементируется до 98 в \_\_except функции main.
* **Ожидаемый Counter:** 98.

**Test 12:**

* **Цель:** \_\_try/\_\_except внутри другого \_\_try/\_\_except.
* **Механизм:** Внешний \_\_try: foo1(STATUS\_ACCESS\_VIOLATION). Внешний \_\_except выполняется, Counter = 1. Внутренний \_\_try: foo1(STATUS\_SUCCESS). Внутренний \_\_except выполняется, Counter = 2.
* **Ожидаемый Counter:** 2.

**Test 13:**

* **Цель:** Аналогично Test 12, но с foo2 (доступ к \*BadAddress) во внешнем \_\_try.
* **Механизм:** Аналогично Test 12. Counter становится 1, затем 2.
* **Ожидаемый Counter:** 2.

**Test 14:**

* **Цель:** goto из блока \_\_except наружу, проходя через \_\_finally.
* **Механизм:** Внутренний \_\_try: foo1 вызывает исключение. Внутренний \_\_except выполняется, Counter = 1, goto t9;. Внешний \_\_finally выполняется (т.к. goto покидает его область \_\_try), Counter = 2.
* **Ожидаемый Counter:** 2.

**Test 15:**

* **Цель:** goto из блока \_\_finally наружу, проходя через другой \_\_finally. (Пропускается, если BAIL\_IN\_FINALLY не определен).
* **Механизм (если выполняется):** Внутренний \_\_try: Counter = 1. Внутренний \_\_finally: Counter = 2, goto t10;. Внешний \_\_finally выполняется, Counter = 3.
* **Ожидаемый Counter (если выполняется):** 3.

**Test 16:**

* **Цель:** goto из \_\_except в \_\_finally того же уровня вложенности. (Пропускается, если BAIL\_IN\_FINALLY не определен).
* **Механизм (если выполняется):** Глубоко вложенный \_\_try: Counter = 1, foo1 вызывает исключение. \_\_except на этом уровне: Counter = 2, goto t11;. \_\_finally уровнем выше: Counter = 3. Метка t11 находится внутри этого \_\_finally. Внешний \_\_finally: Counter = 4.
* **Ожидаемый Counter (если выполняется):** 4.

**Test 17:**

* **Цель:** goto из \_\_finally в \_\_finally того же уровня. (Пропускается, если BAIL\_IN\_FINALLY не определен).
* **Механизм (если выполняется):** Внутренний \_\_try: Counter = 1. Внутренний \_\_finally: Counter = 2, goto t12;. Метка t12 внутри этого же \_\_finally. Внешний \_\_finally: Counter = 3.
* **Ожидаемый Counter (если выполняется):** 3.

**Test 18:**

* **Цель:** return из \_\_except (в функции eret), проходя через \_\_finally.
* **Механизм:** В main: Counter = 1. Вызывается eret. В eret: foo1 вызывает исключение. \_\_except в eret выполняется, \*Counter (из main) становится 2, return;. \_\_finally в eret выполняется, \*Counter становится 3. Возврат в main. \_\_finally в main выполняется, Counter становится 4.
* **Ожидаемый Counter:** 4.

**Test 19:**

* **Цель:** return из \_\_finally (в функции fret). (Пропускается, если BAIL\_IN\_FINALLY не определен).
* **Механизм (если выполняется):** В main: Counter = 1. Вызывается fret. В fret: первый \_\_finally: \*Counter (из main) становится 2, return;. Второй (вложенный) \_\_finally в fret: \*Counter становится 3. Возврат в main. \_\_finally в main выполняется, Counter становится 4 (ошибка в описании теста в коде, должно быть 4, если Counter инкрементируется на 1 в каждом finally до return. Если fret имеет структуру try { try {} finally { C+=1; return; } } finally { C+=1; }, то будет 1(main) + 1(fret\_inner\_fin) + 1(fret\_outer\_fin) + 1(main\_fin) = 4. Однако, код fret немного другой: try { try { C+=1; } finally { C+=1; return; } } finally { C+=1; }. Тогда: main\_C=1. fret\_C=1 (try). fret\_C=2 (inner\_fin), return. fret\_outer\_fin *не выполнится* из-за return. main\_fin C=3. Если main\_C изначально 0: fret\_C=1 (try). fret\_C=2 (inner\_fin), return. main\_C=0 (init), main\_C=1 (try). fret вызвана, fret\_C (переданный как &Counter) становится 1, потом 2, return. main\_fin выполнится (т.к. fret вернула управление), Counter станет 3. Тест ожидает 5, что странно без детального пошагового прогона fret.
  + fret: Counter (из main) = 0. В fret: \_\_try { \_\_try { \*Counter += 1; (C=1) } \_\_finally { \*Counter += 1; (C=2) return; } } \_\_finally { \*Counter += 1; }.
  + main: Counter = 0. \_\_try { Counter += 1; (C=1) fret(&Counter); } \_\_finally { Counter += 1; }.
  + Вызов fret(&Counter): Counter (в fret) указывает на main::Counter.
  + fret: \*Counter (т.е. main::Counter) становится 1+1=2. return. Внешний \_\_finally в fret не выполняется.
  + main: fret вернула. \_\_finally в main: Counter (который был 2) становится 3.
  + **Ожидаемый Counter (если выполняется и логика верна):** 3. В коде ожидается 5, что указывает на более сложную семантику return из \_\_finally или ошибку в тесте/моем понимании. Типично return из \_\_finally считается плохой практикой и может вести к "потере" исходного исключения, если оно было.

**Test 20:**

* **Цель:** Простой setjmp/longjmp.
* **Механизм:** setjmp возвращает 0: Counter = 1. longjmp. setjmp возвращает 1 (значение из longjmp): Counter = 2.
* **Ожидаемый Counter:** 2.

**Test 21:**

* **Цель:** longjmp из \_\_finally, который выполняется последовательно.
* **Механизм:** setjmp возвращает 0. \_\_try: Counter = 1. \_\_finally: Counter = 2, longjmp. setjmp возвращает 1: Counter = 3.
* **Ожидаемый Counter:** 3.

**Test 22:**

* **Цель:** setjmp внутри \_\_try, longjmp из \_\_finally.
* **Механизм:** \_\_try: setjmp возвращает 0, Counter = 1. \_\_finally: Counter = 2. Условие if (Counter == 2) истинно, Counter = 3, longjmp. \_\_try: setjmp теперь возвращает 1, Counter (был 3) становится 4. \_\_finally (снова): Counter (был 4) становится 5. Условие if ложно.
* **Ожидаемый Counter:** 5.

**Test 23:**

* **Цель:** "Столкновение раскруток" (collided unwind). longjmp из \_\_finally во время обработки исключения.
* **Механизм:** setjmp возвращает 0. Внешний \_\_try: внутренний \_\_try: Counter = 1, RaiseStatus. Внутренний \_\_finally: Counter = 2, longjmp. longjmp прерывает обработку исключения. setjmp возвращает 1: Counter (был 2) становится 3. Внешний \_\_except не выполняется.
* **Ожидаемый Counter:** 3.

**Test 24:**

* **Цель:** Более глубокое столкновение раскруток с несколькими \_\_finally.
* **Механизм:** setjmp возвращает 0. Глубоко вложенный \_\_try: Counter = 1, RaiseStatus. Первый \_\_finally (самый внутренний): Counter = 2. Второй \_\_finally: Counter = 3, longjmp. longjmp прерывает обработку исключения. setjmp возвращает 1: Counter (был 3) становится 4. Третий \_\_finally (самый внешний в цепочке до \_\_except) выполняется, Counter (был 4) становится 5. Внешний \_\_except не выполняется.
* **Ожидаемый Counter:** 5.

**Test 25:**

* **Цель:** longjmp из функции dojump, которая имеет вложенные \_\_finally.
* **Механизм:** setjmp возвращает 0. dojump вызывается:
  + dojump: \*Counter (ссылается на main::Counter) инкрементируется до 1. RaiseStatus. Первый \_\_finally в dojump: \*Counter = 2. Второй \_\_finally в dojump: \*Counter = 3, longjmp.
  + longjmp прерывает обработку исключения.
  + setjmp в main возвращает 1: Counter (был 3) становится 4.
  + Первый \_\_finally в main (после dojump): Counter = 5.
  + Второй \_\_finally в main: Counter = 6.
  + Внешний \_\_except в main: не выполняется.
  + Counter (был 6) + 1 в конце if (setjmp...) else { Counter += 1}: Counter = 7 (если longjmp возвращает 1, и Counter был 6 до else блока).
  + Let's trace:
    1. main: Counter = 0. setjmp returns 0.
    2. main: \_\_try { \_\_try { \_\_try { Counter += 1 (C=1); dojump(JumpBuffer, &Counter); } \_\_finally { Counter += 1 (C=X+1); } } \_\_finally { Counter += 1 (C=Y+1); } } \_\_except { Counter += 1; }.
    3. dojump(&Counter): Counter is main::Counter.  
       \_\_try { \_\_try { \*Counter += 1 (C=1+1=2); RaiseStatus; } \_\_finally { \*Counter += 1 (C=2+1=3); } } \_\_finally { \*Counter += 1 (C=3+1=4); longjmp(JumpBuffer, 1); }.
    4. longjmp occurs. main::Counter is 4.
    5. main: setjmp returns 1. else block: Counter += 1 (C=4+1=5).  
       The \_\_finally blocks in main associated with the setjmp(0) path are *not* re-executed on the longjmp path. The unwind from longjmp *does* execute \_\_finally blocks up to the setjmp location.  
       The \_\_finally blocks in dojump *are* executed during the longjmp unwind from within dojump.  
       So, when longjmp happens from dojump, main::Counter is 4. setjmp returns 1. main::Counter becomes 5.  
       Test expects 7. This suggests the \_\_finally blocks in main *are* executed as part of the longjmp unwind.  
       If longjmp from dojump unwinds main's \_\_finally blocks:  
       dojump makes Counter = 4 then longjmp.  
       Unwind from longjmp:
    6. \_\_finally { Counter += 1; } (inner in main): Counter = 5
    7. \_\_finally { Counter += 1; } (outer in main): Counter = 6  
       Then setjmp returns 1, else { Counter +=1; }: Counter = 7.  
       This is consistent with longjmp unwinding \_\_finally blocks.
* **Ожидаемый Counter:** 7.

**Test 26:**

* **Цель:** Похоже на Test 25, но longjmp из \_\_finally уровнем выше в main.
* **Механизм:** setjmp возвращает 0.
  + dojump вызывается: \*Counter (из main) = 1 (в main) -> 2 (в dojump) -> RaiseStatus -> 3 (1й finally в dojump) -> 4 (2й finally в dojump, longjmp).
  + longjmp из dojump перехватывается setjmp в main. Counter = 4.
  + \_\_finally (вокруг dojump): Counter = 5.
  + \_\_finally (следующий, который содержит longjmp): Counter = 6, longjmp(JumpBuffer, 1).
  + Этот longjmp снова перехватывается тем же setjmp. Counter = 6.
  + setjmp возвращает 1 (от второго longjmp). else блок: Counter (был 6) становится 7.
  + Внешний \_\_finally в main: Counter (был 7) становится 8.
  + \_\_except не выполняется.
* **Ожидаемый Counter:** 8.

**Test 27:**

* **Цель:** Вложенные исключения. except1 вызывает except3, который вызывает RaiseStatus, а затем except3 снова вызывает RaiseStatus в своем фильтре.
* **Механизм:**
  1. main: Counter = 0. Внешний \_\_try.
  2. main: Внутренний \_\_try: Counter += 1 (C=1). except1(&Counter):
     + except1: \*Counter += 5 (C=1+5=6). RaiseStatus(STATUS\_INTEGER\_OVERFLOW).
     + except1: \_\_except(except3(GetExceptionInformation(), Counter)). Вызов except3:
       - except3: Код STATUS\_INTEGER\_OVERFLOW. \*Counter += 17 (C=6+17=23). RaiseStatus(STATUS\_UNSUCCESSFUL). Это *новое* исключение.
       - Обработчик для STATUS\_UNSUCCESSFUL (внутри except3? Нет, это новое исключение распространяется).
       - Поскольку except3 вызвал RaiseStatus в *фильтре* except1, это исключение (STATUS\_UNSUCCESSFUL) распространяется выше.
       - except1 не завершил свой \_\_except фильтр. Исключение STATUS\_UNSUCCESSFUL выходит из except1.
  3. main: Внутренний \_\_except(except2(GetExceptionInformation(), Counter)). Вызов except2:
     + except2: Код STATUS\_UNSUCCESSFUL. Флаг EXCEPTION\_NESTED\_CALL *не* установлен (это первое исключение на этом уровне). \*Counter += 11 (C=23+11=34). Возвращает EXCEPTION\_EXECUTE\_HANDLER.
  4. main: Внутренний \_\_except блок выполняется: Counter += 2 (C=34+2=36).
  5. Внешний \_\_except не выполняется.
  6. Ожидается 55. Мой подсчет 36. Давайте пересмотрим except3:
     + except3(exc\_info\_overflow, &Counter):
       - ExceptionCode == STATUS\_INTEGER\_OVERFLOW -> \*Counter += 17 (C=23). RaiseStatus(STATUS\_UNSUCCESSFUL).
       - Это RaiseStatus теперь "перезаписывает" текущее исключение. Раскрутка стека из-за STATUS\_UNSUCCESSFUL начинается *изнутри* фильтра except3.
       - except3 вернет EXCEPTION\_EXECUTE\_HANDLER (т.к. \*Counter += 23 не будет достигнут).
     + Фильтр except1 получил EXCEPTION\_EXECUTE\_HANDLER (от except3).
     + Блок \_\_except в except1 выполняется: \*Counter += 7 (C=23+7=30).
     + except1 завершается: \*Counter += 9 (C=30+9=39).
  7. Теперь Counter = 39. Внутренний \_\_try в main завершился нормально (т.к. except1 обработал исключение). Внутренний \_\_except в main не выполняется.
  8. Итого: 39. Все еще не 55.
  9. Проблема в "nested call". Когда except3 вызывает RaiseStatus, это *новое* исключение. Оно ищет обработчик.
     + Если except3 был вызван как фильтр для except1's \_\_except, то новое исключение (STATUS\_UNSUCCESSFUL) будет искать обработчик выше except1.
     + То есть, оно попадет в main's inner \_\_except(except2(...)).
     + except2 получит STATUS\_UNSUCCESSFUL. \*Counter += 11 (C=23+11=34). Вернет EXECUTE\_HANDLER.
     + Блок inner \_\_except в main: Counter += 2 (C=34+2=36).
  10. Все еще 36.
  11. Может быть, EXCEPTION\_NESTED\_CALL флаг устанавливается, когда исключение возникает *внутри* фильтра или обработчика.
  12. except1: C=0 -> C=1 (в main). C=1+5=6. Raise(INT\_OVERFLOW).
      + Filter except3(INT\_OVERFLOW, &C):
        - C=6+17=23. Raise(UNSUCCESSFUL). Это новое исключение.
        - except3 не возвращает значение, т.к. новое исключение прерывает его.
      + UNSUCCESSFUL распространяется. Попадает во внутренний \_\_except(except2(UNSUCCESSFUL, &C)) в main.
        - Filter except2(UNSUCCESSFUL, &C):
          * ExceptionFlags & EXCEPTION\_NESTED\_CALL == 0? Да, т.к. UNSUCCESSFUL не возникло в обработчике UNSUCCESSFUL.
          * C=23+11=34. return EXCEPTION\_EXECUTE\_HANDLER.
        - Блок \_\_except для внутреннего try в main: C=34+2=36.
  13. Ожидание 55 указывает на очень специфическое поведение флага EXCEPTION\_NESTED\_CALL и того, как фильтры взаимодействуют.
  14. Возможно, except3 все-таки возвращает значение, а RaiseStatus в нем рассматривается как часть его логики.
      + except1: C=6. Raise(OVERFLOW). Фильтр except3(OVERFLOW, &C):
        - C=6+17=23. Raise(UNSUCCESSFUL). Допустим, это исключение обрабатывается *внутри* except3 неявно (или тест предполагает это).
        - Если UNSUCCESSFUL было бы обработано и except3 продолжил, то дальше else if. Но UNSUCCESSFUL уже было поднято.
        - Если RaiseStatus в фильтре означает, что фильтр возвращает EXCEPTION\_CONTINUE\_SEARCH для исходного исключения, а новое исключение (UNSUCCESSFUL) становится главным.
        - Тогда OVERFLOW продолжает поиск и попадает во внешний \_\_except в main.
          * Фильтр except2(OVERFLOW, &C): C=23+13=36. return EXCEPTION\_CONTINUE\_SEARCH.
        - Внешний \_\_except в main (самый внешний): C=36+3=39.
  15. Это очень сложный тест. Значение 55 получается, если:
      + main: C=0 -> C=1.
      + except1(&C): C=1+5=6. Raise(OVERFLOW).
      + Фильтр except1 -> except3(OVERFLOW, &C):
        - C=6+17=23. Raise(UNSUCCESSFUL).
        - Это UNSUCCESSFUL перехватывается *самим except3* во второй ветке else if (т.к. теперь ExceptionCode == UNSUCCESSFUL и NESTED\_CALL установлен):
          * C=23+19=42. return EXCEPTION\_CONTINUE\_SEARCH (для UNSUCCESSFUL).
        - Этот CONTINUE\_SEARCH (для UNSUCCESSFUL) означает, что except3 не обработал UNSUCCESSFUL.
        - Но except3 должен вернуть значение для исходного OVERFLOW. Что он вернет? Последний \*Counter += 23; return EXCEPTION\_EXECUTE\_HANDLER; не достигается.
        - Если RaiseStatus в фильтре приводит к тому, что фильтр немедленно возвращает EXCEPTION\_CONTINUE\_SEARCH для исходного исключения, и новое исключение становится доминирующим.
        - Тогда OVERFLOW не обрабатывается except1. UNSUCCESSFUL (с C=23) выходит из except1.
        - main's inner \_\_except(except2(UNSUCCESSFUL, &C)):
          * except2: C=23+11=34. return EXECUTE\_HANDLER.
          * Блок inner \_\_except: C=34+2=36.
  16. **Правильный путь к 55**:
      + main: Counter = 0. \_\_try { Counter += 1; (C=1) except1(&Counter); } ...
      + except1: \*Counter += 5; (C=6) RaiseStatus(OVERFLOW);
      + except1: \_\_except(except3(GetExceptionInformation() /\*OVERFLOW\*/, Counter))
        - except3(OVERFLOW, &C):
          * ExceptionCode == OVERFLOW -> \*Counter += 17; (C=6+17=23).
          * RaiseStatus(UNSUCCESSFUL); // Это новое, вложенное исключение.
          * Раскрутка стека для UNSUCCESSFUL начинается из except3.
          * except3 сам становится целью для этого UNSUCCESSFUL.
          * except3(UNSUCCESSFUL, &C) (рекурсивный вызов диспетчера, флаг NESTED\_CALL теперь установлен для UNSUCCESSFUL):

ExceptionCode == UNSUCCESSFUL and NESTED\_CALL is true -> \*Counter += 19; (C=23+19=42).

return EXCEPTION\_CONTINUE\_SEARCH; (для UNSUCCESSFUL).

* + - * + UNSUCCESSFUL не обработано except3. Оно распространяется дальше.
        + except3 должен был вернуть значение для OVERFLOW. Поскольку UNSUCCESSFUL прервало его, поведение не определено стандартом, но компилятор MSVC может сделать так, что except3 вернет последнее вычисленное значение перед RaiseStatus или EXCEPTION\_CONTINUE\_SEARCH.
        + Предположим, except3 вернул EXCEPTION\_EXECUTE\_HANDLER для OVERFLOW из-за \*Counter += 23 ветки, которая не была достигнута, но может быть поведением по умолчанию при прерывании. **Или, более вероятно,** RaiseStatus в фильтре приводит к тому, что фильтр возвращает EXCEPTION\_CONTINUE\_SEARCH для первоначального исключения (OVERFLOW).
    - Если except3 вернул CONTINUE\_SEARCH для OVERFLOW, то except1 не обрабатывает OVERFLOW.  
      OVERFLOW теперь ищет обработчик выше except1. UNSUCCESSFUL также ищет обработчик выше except1. Обычно новое исключение имеет приоритет.
    - UNSUCCESSFUL (с C=42) достигает main's inner \_\_except(except2(GetExceptionInformation() /\*UNSUCCESSFUL\*/, Counter)).
      * except2(UNSUCCESSFUL, &C):
        + ExceptionCode == UNSUCCESSFUL and NESTED\_CALL is false (т.к. UNSUCCESSFUL не возникло в обработчике UNSUCCESSFUL). -> \*Counter += 11; (C=42+11=53).
        + return EXCEPTION\_EXECUTE\_HANDLER; (для UNSUCCESSFUL).
      * Блок inner \_\_except в main выполняется: Counter += 2; (C=53+2=55).
    - Внешний \_\_except в main не выполняется.
* **Ожидаемый Counter:** 55 (очень сложный сценарий).

**Test 28:**

* **Цель:** Исключение целочисленного переполнения (STATUS\_INTEGER\_OVERFLOW).
* **Механизм:** Counter = 1. addtwo вызывает RaiseStatus(STATUS\_INTEGER\_OVERFLOW). \_\_except обрабатывает. Counter = 2.
* **Ожидаемый Counter:** 2.

**Test 29:**

* **Цель:** Исключение из-за невыровненных данных (STATUS\_DATATYPE\_MISALIGNMENT). (Пропускается, если DO\_ALIGNMENT\_TEST не определен, обычно для ARM/ARM64).
* **Механизм (если выполняется):** Counter = 1. foo2 с BadByte (нечетный адрес) и Interlock = TRUE вызывает исключение. \_\_except обрабатывает. Counter = 2.
* **Ожидаемый Counter (если выполняется):** 2.

**Test 30-50:** Касаются continue, break, \_\_leave в циклах и switch в сочетании с \_\_try/\_\_except/\_\_finally.  
\* **continue из \_\_try**: \_\_finally выполняется.  
\* **break из \_\_try**: \_\_finally выполняется.  
\* **\_\_leave из \_\_try**: \_\_finally выполняется, abnormal\_termination() = FALSE.  
\* Если continue/break из \_\_finally (требует BAIL\_IN\_FINALLY), это особый случай.

**Test 30:** continue из \_\_try в цикле с \_\_except.

* **Механизм:** Если Index1 четное, continue. Нечетное: Counter += 1, Counter += 2. Цикл 5 раз для нечетных.
* **Ожидаемый Counter:** 5 \* (1+2) = 15.

**Test 31:** continue из \_\_try в цикле с \_\_finally.

* **Механизм:** Если Index1 четное, continue, но \_\_finally выполнится (Counter += 2). Нечетное: Counter += 1 (try), Counter += 2 (finally), Counter += 3 (после try-finally).
  + Четные (5 раз): Counter += 2. Итого 10.
  + Нечетные (5 раз): Counter += 1+2+3 = 6. Итого 30.
* **Ожидаемый Counter:** 10 + 30 = 40.

**Test 32:** continue из вложенного \_\_try в цикле с \_\_except.

* **Механизм:** Если Index1 четное, continue. Нечетное: Counter += 1 (inner try), Counter += 2 (after inner try-except), Counter += 3 (after outer try-except).
* **Ожидаемый Counter:** 5 \* (1+2+3) = 30.

**Test 33:** continue из вложенного \_\_try в цикле с \_\_finally.

* **Механизм:** Четные: continue -> inner \_\_finally (C+=2) -> outer \_\_finally (C+=4). Итого 5 \* (2+4) = 30.  
  Нечетные: C+=1 (inner try) -> inner \_\_finally (C+=2) -> C+=3 (after inner) -> outer \_\_finally (C+=4) -> C+=5 (after outer). Итого 5 \* (1+2+3+4+5) = 5 \* 15 = 75.
* **Ожидаемый Counter:** 30 + 75 = 105.

**Test 34-36:** continue из \_\_finally. (Требует BAIL\_IN\_FINALLY).

* **Test 34:** Counter = 25.
* **Test 35:** Counter = 75.
* **Test 36:** Counter = 115.

**Test 37-43:** break из \_\_try или \_\_finally.

* **Test 37:** break из \_\_try с \_\_except. Index1=0: C+=1, C+=2. Index1=1: break. C=3.
* **Ожидаемый Counter:** 3.
* **Test 38:** break из \_\_try с \_\_finally. Index1=0: C+=1 (try), C+=2 (fin), C+=3 (after). Index1=1: break -> C+=2 (fin).
  + Index1=0: C=1+2+3=6. Index1=1: C (был 6) +=2 (fin от break). C=8.
* **Ожидаемый Counter:** 8.
* Остальные аналогичны, с учетом вложенности и BAIL\_IN\_FINALLY.

**Test 44-50:** break из \_\_try или \_\_finally внутри switch.

* Поведение break здесь завершает switch, а не только \_\_try. \_\_finally должен выполниться.
* **Test 44:** Counter = 0.
* **Test 45:** Counter = 2.

**Test 51-55:** \_\_leave из \_\_try.

* \_\_leave выходит из \_\_try блока. \_\_finally выполняется, abnormal\_termination() = FALSE.
* **Test 51:** C+=3 (try, before leave). \_\_leave. \_\_finally: abnormal\_termination() = FALSE, C+=5.
* **Ожидаемый Counter:** 3 + 5 = 8.
* **Test 52:** Аналогично, в цикле. C=8.
* **Test 53:** Аналогично, в switch. C=8.
* **Test 54:** Вложенные \_\_leave. Inner \_\_try: C+=3, \_\_leave. Inner \_\_finally: C+=5. C*now*=8. Outer \_\_try: C+=3, \_\_leave. Outer \_\_finally: C+=5.
* **Ожидаемый Counter:** 3 + 5 + 3 + 5 = 16.
* **Test 55:** \_\_leave из \_\_finally. (Требует BAIL\_IN\_FINALLY). C=13.

**Test 56-59:** SEH внутри SEH блоков.

* **Test 56:** \_\_try/\_\_finally внутри \_\_except.
  + C=1. RaiseStatus. \_\_except выполняется: C+=3 (try\_inner), C+=5 (finally\_inner).
* **Ожидаемый Counter:** 1 + 3 + 5 = 9.
* **Test 57:** \_\_try/\_\_finally внутри \_\_finally.
  + C=1 (outer\_try). Outer \_\_finally: C+=3 (inner\_try), C+=5 (inner\_finally).
* **Ожидаемый Counter:** 1 + 3 + 5 = 9.
* **Test 58:** \_\_try/\_\_except внутри \_\_finally. (Требует NEST\_IN\_FINALLY). C=9.
* **Test 59:** \_\_try/\_\_except внутри \_\_except. C=9.

**Test 60:** goto из внутреннего \_\_try наружу, пропуская \_\_except, но вызывая исключение позже.

* **Механизм:** Inner \_\_try: goto outside;. Inner \_\_except не выполняется. Метка outside: RaiseStatus. Outer \_\_except: C+=3.
* **Ожидаемый Counter:** 3.

**Test 61:** Вызов Test61Part2 (с \_\_try/\_\_finally и вложенным \_\_try/\_\_except в \_\_finally). (Требует NEST\_IN\_FINALLY).

* Test61Part2: \*Counter -= 1 (C=-1). RaiseStatus. \_\_finally: \_\_try { \*Counter += 2 (C=1); RaiseStatus; } \_\_except { \*Counter += 5 (C=6); }. \*Counter += 7 (C=13).
* main: \_\_except { \*Counter += 11 (C=24); }.
* **Ожидаемый Counter:** 24.

**Test 62:** Точность исключений с плавающей точкой. (Требует DO\_FLOAT\_EXCEPTION\_TEST).

* **Механизм:** Включается отлов EM\_OVERFLOW. SquareDouble(1.7e300) вызовет переполнение. \_\_except выполнится, C=1. Внутренний SquareDouble(1.0) не вызовет.
* **Ожидаемый Counter:** 1.

**Test 63:** Исключение в \_\_finally внешнего \_\_try/\_\_except.

* **Механизм:** \_\_try { C+=1 (C=1); } \_\_finally { C+=3 (C=4); RaiseStatus; }. Это новое исключение.
* \_\_except(1) { C+=6 (C=10); }.
* **Ожидаемый Counter:** 10.

**Test 64:** Исключение и в \_\_try, и в \_\_finally.

* **Механизм:** \_\_try { C+=1 (C=1); RaiseStatus(EXC1); } \_\_finally { C+=3 (C=4); RaiseStatus(EXC2); }.
* Исключение EXC2 из \_\_finally "затирает" EXC1.
* \_\_except(1) { C+=6 (C=10); }.
* **Ожидаемый Counter:** 10.

**Test 65:** \*BadAddress в \_\_finally. Аналогично Test 63.

* **Ожидаемый Counter:** 10.

**Test 66:** \*BadAddress и в \_\_try, и в \_\_finally. Аналогично Test 64.

* **Ожидаемый Counter:** 10.

**Test 67:** Вложенные \_\_finally, исключение во внутреннем \_\_finally и во внешнем \_\_try.

* **Механизм:** \_\_try { \*BadAddress; } \_\_finally { \_\_try { C=0; } \_\_finally { /\* C!=0 no\*/ } C+=1 (C=1); \*BadAddress; }.
* Исключение из внешнего \_\_finally (второй \*BadAddress) доминирует.
* \_\_except(1) { C+=1 (C=2); }.
* **Ожидаемый Counter:** 2.

**Test 68:** Аналогично Test 67, но с RaiseStatus.

* **Ожидаемый Counter:** 2.

**Patch Guard тесты из xcpt4pg.c (функция PgTests, вызываемая из main)**  
Эти тесты (69-78) проверяют сложные сценарии раскрутки стека, часто с return из \_\_finally или вызовом фильтра из \_\_finally. Переменная State это Counter, Fault это BadAddress.

**Test 69:**

* **Механизм:** PgTest69 вызывается с Counter=0.
  + PgTest69: \*Fault += 1 (AV). \_\_finally: AbnormalTermination() да. \*State == 1 нет. \*Fault += 1.
  + \_\_except(((\*State += 1) == 1) ? PgFilter() : ...): \*State (Counter) становится 1. PgFilter() вызывается: printf, return EXECUTE\_HANDLER.
  + Блок \_\_except: \*State == 2 нет. \*Fault += 1.
  + В PgTest69 Counter становится 1.
  + В main после PgTest69 Counter остался 1. Ожидается 2.
  + Если (\*State += 1) выполняется, и \*State становится 1. Фильтр PgFilter вызывается. Counter становится 2 из-за (\*State += 1) во внешнем \_\_except в PgTest69.
* **Ожидаемый Counter:** 2.

**Test 70-78:** Пропускаются, если BAIL\_IN\_FINALLY не определен. Они тестируют return или вызов фильтра из \_\_finally в разных комбинациях вложенности.

* **Test 70:** Counter=2.
* **Test 71:** Counter=9.
* **Test 72:** Counter=12.
* **Test 73:** Counter=15.
* **Test 74:** Counter=18.
* **Test 75:** Counter=35.
* **Test 76:** Counter=40.
* **Test 77:** Counter=45.
* **Test 78:** Counter=50.

**Extended/External тесты из xcpt4ex.c (функция ExTests, вызываемая из main)**  
Здесь функции TestNN определены в xcpt4ex.c или xcpt4cxx.cpp.

**Test 79:**

* **Механизм:** Test79 вызывается с Counter=0.
  + Test79: \*Fault += 1 (AV). Внутренний \_\_finally: printf, \*Fault += 1. Средний \_\_finally: printf.
  + \_\_except(\*Counter += 1, printf("filter 1..."), EXCEPTION\_CONTINUE\_SEARCH): \*Counter становится 1. Возвращает CONTINUE\_SEARCH.
  + Исключение (AV) не обработано в Test79. Оно распространяется в main.
  + main: \_\_except(printf("filter 2..."), EXCEPTION\_EXECUTE\_HANDLER): Counter (который 1) += 1. Становится 2.
* Ожидалось 3. Значит \*Counter +=1 в фильтре Test79 выполнилось, а потом Counter +=1 в main в \_\_except. Где еще +1?
  + Test79: \*Counter (main's Counter) = 0. \*Fault (AV). Первый finally. Второй finally. \_\_except фильтр: \*Counter += 1 (main's C=1), CONTINUE\_SEARCH.
  + main: Test79 вызвала AV. \_\_except фильтр: printf, EXECUTE\_HANDLER. Блок \_\_except: Counter += 1 (main's C=1+1=2).
  + Чтобы было 3, нужно, чтобы Counter в Test79 был отдельным, а не ссылкой. Но он PLONG.
  + Если в main: Counter = 0. Test79(&Counter, BadAddress). В Test79 фильтр \*Counter += 1 (main::Counter стал 1). Вернул CONTINUE\_SEARCH.
  + В main \_\_except блок: Counter += 1 (main::Counter стал 2).
  + **В Test79 сам Counter не используется, а \*Counter - это main::Counter. Ожидание 3 - ошибка в тесте или я что-то упускаю.**
  + А, main в xcpt4u.c, а ExTests вызывает Test79.
  + В ExTests: Counter=0. Вызов Test79(&Counter, BadAddress).
    - Test79: \*Counter (т.е. ExTests::Counter) становится 1 из-за фильтра. Исключение продолжает поиск.
    - ExTests: \_\_except { Counter += 1; }. ExTests::Counter становится 1+1=2.
    - printf в ExTests: if (Counter == 3). Ошибка в тесте, должно быть 2.

**Test 80:**

* **Механизм:** Test80 использует глобальную G.
  + G=1. while(G) (true). \_\_try { if (G==10) return 1; if (G==1) continue; } \_\_finally { G=0; }.
  + Итерация 1: G=1. continue. \_\_finally: G=0. while(G) (false).
  + \_\_finally (внешний): G=10. return 0;.
* **Ожидаемый результат:** 0. (passed)

**Test 81:**

* **Механизм:** Test81(&Counter). Counter (из ExTests) = 0.
  + \*AvPtr = '\0' (AV). \_\_except(EXECUTE\_HANDLER) { \_\_leave; }.
  + \_\_leave выходит из \_\_except в конец \_\_try блока, который содержит \_\_except.
  + \_\_finally { \*Counter += 1; }. Counter (из ExTests) становится 1.
* **Ожидаемый Counter:** 1. (passed)

**Test 82:** (Функция Test82 в xcpt4cxx.cpp, Test82Foo там же)

* **Механизм:** Counter (из ExTests) = 1. Test82(&Counter).
  + Test82Foo(): throw 1; (C++ исключение).
  + Раскрутка стека из-за C++ throw. \_\_finally в Test82 выполняется:
    - abnormal\_termination() = TRUE. switch(\*Counter) (который 1). case 1: retval = 0; break;.
  + \_\_except(1) в Test82 *не перехватит* C++ исключение с /EHsc. C++ исключение 1 распространится из Test82.
  + В Test82: \*Counter = retval; (т.е. \*Counter = 0).
  + Функция Test82 завершается (из-за C++ исключения, которое не поймано локально, но перед этим \*Counter был изменен).
  + В ExTests C++ исключение не ловится и программа, вероятно, завершится, *если бы не было других обработчиков выше*. Но тест проверяет значение Counter.
* **Ожидаемый Counter:** 0. (passed)

**Test 83:**

* **Механизм:** Похож на Test 80, но с \_\_except вокруг while.
  + G=1. while(G) (true). \_\_try { ... if (G==1) continue; } \_\_finally { G=0; }.
  + Итерация 1: G=1. continue. \_\_finally: G=0. while(G) (false).
  + \_\_except не выполняется. \_\_finally (внешний): G=10. return 0;.
* **Ожидаемый результат:** 0. (passed)

**Test 84:**

* **Механизм:** Counter (из ExTests) = 0. Test84(&Counter).
  + \*Fault += 1 (AV). \_\_except(EXECUTE\_HANDLER) { \_\_try { return; } \_\_finally { \*Counter += 1; } }.
  + При return из \_\_except, вложенный \_\_finally выполняется. \*Counter становится 1.
  + Внешний \_\_finally: AbnormalTermination() = TRUE (т.к. return был из \_\_except, что является частью \_\_try). \*Counter += 1. \*Counter становится 2.
* **Ожидаемый Counter:** 2. (passed)

**Test 85:** Очень сложный, много вложенных \_\_try/\_\_finally/\_\_except и цикл.

* **Ожидаемый Counter:** 7. (passed)

**Test 86:** Похож на Test 84, но глубже вложенность.

* **Механизм:** Counter=0.
  + \*Fault (AV). \_\_except { \_\_try {return;} \_\_finally {\*C+=1 (C=1);} }.
  + Outer \_\_finally {\*C+=1 (C=2);}.
  + Next outer \_\_except (не достигнут).
  + Next outer \_\_finally {\*C+=1 (C=3);}.
  + Next outer \_\_except (не достигнут).
  + Outermost \_\_finally {\*C+=1 (C=4);}.
* **Ожидаемый Counter:** 4. (passed)

**Test 87, 88, 89:** (Пропускаются если BAIL\_IN\_FINALLY не определен или для x86). Сложные тесты с return или выходами из \_\_finally.

* **Test 87:** Counter=104.
* **Test 88:** Counter=6.
* **Test 89:** Counter=69.

**Test 90:** (Функция Test90 и CxxLongjump90 в xcpt4cxx.cpp)

* **Цель:** Взаимодействие setjmp/longjmp с деструкторами C++ объектов.
* **Механизм:** /EHsc включен, так что деструкторы должны вызываться при longjmp.
  + Destructable<90>::alive = 0.
  + CxxLongjump90():
    - Destructable<90> l0(0); (alive=1).
    - setjmp(env) возвращает 0.
    - Destructable<90> l1(1); (alive=2).
    - longjmp(env, 1);
      * Деструктор ~l1() вызывается (alive=1).
      * Деструктор ~l0() вызывается (alive=0).
    - setjmp(env) теперь возвращает 1.
    - Destructable<90> l2(2); (alive=1).
    - CxxLongjump90 возвращается.
    - Деструктор ~l2() вызывается (alive=0).
  + Test90: \*State = Destructable<90>::alive;.
* **Ожидаемый Counter (State):** 0. (passed)

Это очень подробный набор тестов, покрывающий множество краевых случаев SEH и его взаимодействия с C++. Некоторые из них, особенно с BAIL\_IN\_FINALLY или сложной вложенностью исключений, проверяют поведение, которое может зависеть от конкретной версии компилятора и архитектуры. Основная идея - убедиться, что раскрутка стека и выполнение \_\_finally блоков происходит корректно в самых разных ситуациях.