Обработка исключений

Основное внимание в данной главе сфокусировано на структурной обработке исключений (Structured Exception Handling, SEH), но наряду с этим обсуждены также обработчики управляющих сигналов консоли и векторная обработка исключений (Vectored Exception Handling, VEH).

SEH предоставляет механизм обеспечения надежности программ, благодаря которому приложения получают возможность реагировать на такие непредсказуемые события, как исключения адресации, арифметические сбои и системные ошибки. Использование SEH позволяет программам осуществлять корректный выход из любой точки программного блока и автоматически выполнять предусмотренную программистом обработку ошибок для восстановления своей работоспособности. SEH гарантирует своевременное освобождение ресурсов и выполнение любых других операций очистки, прежде чем блок, поток или процесс закончат работу либо под управлением программы, либо в ответ на возникновение исключительной ситуации. Кроме того, SEH легко добавляется в существующие программные коды, во многих случаях обеспечивая упрощение логики работы программы.

Мы используем SEH в приведенных ниже примерах программ и расширим посредством этого механизма возможности функции обработки ошибок ReportError, которая была введена в главе 2. Обычно сфера применимости SEH ограничивается программами, написанными на языке на C. Вместе с тем, представленные ниже возможности SEH воспроизводятся в C++, C# и других языках программирования с использованием весьма похожих механизмов.

В настоящей главе описаны также обработчики управляющих сигналов консоли, благодаря которым программы могут воспринимать внешние сигналы, вырабатываемые, например, при нажатии сочетания клавиш <Ctrl+C>, выходе пользователя из системы или завершении работы системы. Кроме того, использование подобных сигналов обеспечивает реализацию ограниченных форм межпроцессного взаимодействия.

Глава завершается рассмотрением векторной обработки исключений, которая потребует от вас использования операционных систем Windows XP или Windows Server 2003. Благодаря VEH пользователь получает возможность определить функции, которые должны вызываться сразу же после возникновения исключения, не дожидаясь активизации SEH.

Исключения и обработчики исключений

В отсутствие обработки исключений возникновение любой нестандартной ситуации, например, попытки разыменования нулевого указателя или деления на ноль, приведет к немедленному прекращению выполнения программы. В качестве примера, иллюстрирующего проблемы, которые могут при этом возникать, можно назвать создаваемые программой

временные файлы, подлежащие удалению до того, как программа завершит свою работу. SEH предоставляет возможность определить блок программного кода, или *обработчик исключений* (exception handler), который в случае возникновения исключения удалит временные файлы.

Поддержка SEH обеспечивается за счет совместного использования функций Windows, средств поддержки языков программирования, предоставляемых компилятором, и средств поддержки времени выполнения. Какой именно язык программирования поддерживается, зависит от конкретной системы; наши примеры ориентированы на Microsoft C.

Блоки try и except

Все начинается с выяснения того, в каких именно блоках программного кода вы намерены контролировать возникновение нестандартных ситуаций, после чего этим блокам должны быть предоставлены обработчики исключений в соответствии с приведенным ниже описанием. Можно контролировать как функцию в целом, так и ее отдельные программные блоки или подфункции, предусмотрев для них независимые обработчики исключений.

Ниже перечислены характерные признаки участков программного кода, для которых целесообразно предусматривать отдельные обработчики исключений.

• Возможность возникновения регистрируемых ошибок, включая ошибки системных вызовов, в условиях, когда необходимо организовать устранение последствий ошибки, а не предоставлять программе возможность прекращения выполнения.

• Интенсивное использование указателей, повышающее вероятность попыток разыменования указателей, инициализация которых не была выполнена должным образом.

• Интенсивное использование данных в виде массивов, что может сопровождаться выходом значений индексов элементов массива за границы допустимого диапазона.

• В программе выполняются арифметические операции с участием вещественных чисел (чисел с плавающей точкой), и существует риск того, что могут возникать исключения, связанные с попытками деления на ноль, потерей точности при вычислениях и переполнением.

• Наличие вызовов функций, которые могут генерировать исключения либо программным путем, либо в силу того, что их работоспособность не была достаточно тщательно проверена.

Если при изучении примеров, приведенных в этой главе или книге в целом, вы решите отслеживать исключения, которые могут возникать на том или ином участке программы, создайте для него блоки try и except, как показано ниже:

```
__try {
 /* Блок контролируемого кода */
```

} __ except*(выражение_фильтра)* {
/* Блок обработки исключений */
}

Имейте в виду, что __try и __except — это ключевые слова, распознаваемые компилятором.

Блоки try являются частью обычного кода приложения. Если на данном участке кода возникает исключение, ОС передает управление обработчику исключений, который представляет собой блок программного кода, следующий за ключевым словом_ except. Характер последующих действий определяется значением параметра выражение_фильтра.

Обратите внимание, что исключение может возникнуть также в пределах блока, находящегося внутри try-блока; в этом случае средства поддержки времени исполнения "разворачивают" стек, чтобы отыскать в нем информацию об обработчике исключений, после чего передают управление этому обработчику. То же самое происходит и в тех случаях, когда исключения возникают внутри функций, вызванных в пределах try-блока.

На рис. 4.1 показано, как располагается в стеке информация об обработчике исключений во время возникновения исключения. Как только обработчик исключений завершит свою работу, управление передается оператору, который следует за блоком except, если только в самом обработчике исключений не предусмотрены иные операторы ветвления, изменяющие ход выполнения программы.

Выражения фильтров и их значения

Параметр выражение_фильтра в операторе except вычисляется сразу же после того, как возникает исключение. В качестве выражения может выступать литеральная константа, вызов *функции фильтра* (filter function) или условное выражение. В любом случае выражение должно возвращать одно из следующих трех значений:

1. EXCEPTION_EXECUTE_HANDLER — система выполняет операторы блока обработки исключений, как показано на рис. 4.1 (см. программу 4.1). Это соответствует обычному случаю.
2. EXCEPTION_CONTINUE_SEARCH — система игнорирует данный обработчик исключений и пытается найти обработчик исключений в охватывающем блоке, продолжая этот процесс аналогичным образом до тех пор, пока не будет найден обработчик исключений.
3. EXCEPTION_CONTINUE_EXECUTION — система немедленно возвращает управление в точку, в которой возникло исключение. В случае некоторых исключений дальнейшее выполнение программы невозможно, но если такие попытки делаются, то генерируется повторное исключение.

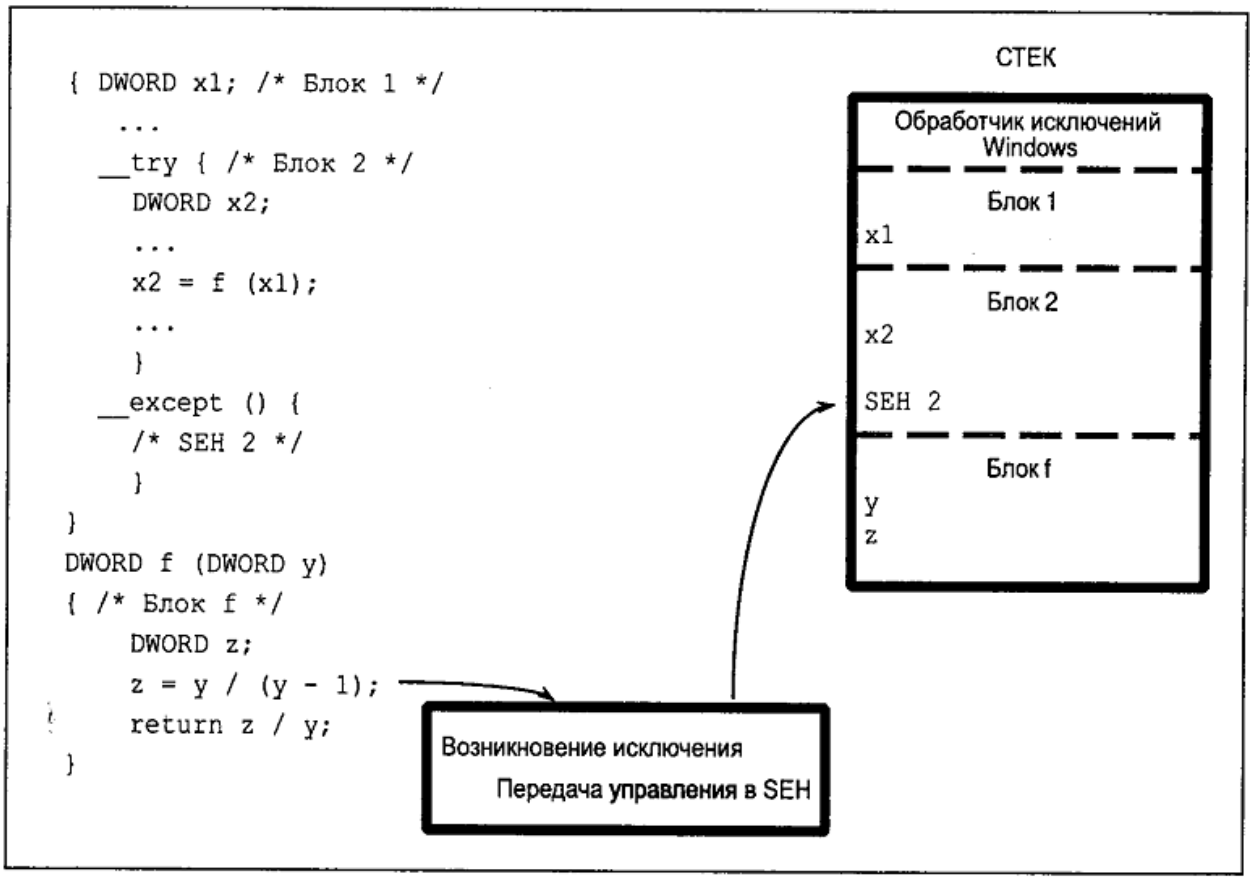

Рис. 4.1. SEH, блоки и функции

Ниже приведен простой пример, в котором обработчик исключений используется для удаления временного файла в тех случаях, когда исключение возникает в теле цикла. Заметьте, что ключевое слово __try может быть применено к любому блоку, включая блоки, связанные с операторами while, if или любым другим оператором ветвления. В данном примере возникновение любого исключения приводит к удалению временного файла и закрытию дескриптора, после чего выполнение цикла возобновляется.

```
GetTempFileName(TempFile, …);
while (…) __try {
 hFile = CreateFile(TempFile, …, OPEN_ALWAYS, …);
 SetFilePointer(hFile, 0, NULL, FILE_END);
 WriteFile(hFile, …);
 i = *p; /* В этом месте программы возможно возникновение исключения адресации. */
 CloseHandle (hFile);
 …
} __except (EXCEPTION_EXECUTE_HANDLER) {
 CloseHandle(hFile);
 DeleteFile(TempFile);
 /* Переход к выполнению очередной итерации цикла. */
}
```

/* Сюда передается управление после нормального завершения цикла.
Каждый раз при возникновении исключения дескриптор временного файла закрывается, а сам файл удаляется. */

Ниже описана логика приведенного выше фрагмента кода.

• На каждой итерации цикла в конце файла добавляются новые данные.

• В случае возникновения исключения во время выполнения итерации цикла все данные, накопленные во временном файле, будут уничтожены, и если еще остались невыполненные итерации, то во временном файле начнут накапливаться новые данные.

• В случае возникновения исключения на последней итерации файл прекращает существование. В любом случае файл будет содержать все данные, сгенерированные после предыдущего исключения.

• В примере отмечена лишь одна точка программы, в которой возможно возникновение исключения, хотя исключения могут возникнуть в любой точке тела цикла.

• Чтобы гарантировать закрытие дескриптора файла, это делается как при выходе из цикла, так и перед началом очередной итерации цикла.

Коды исключений

Для точной идентификации типа возникшего исключения блок исключения или выражение фильтра могут использовать следующую функцию:

DWORD GetExceptionCode(VOID)

Код исключения должен быть получен сразу же после возникновения исключения. Поэтому функция фильтра не может просто вызвать функцию GetExceptionCode (это ограничение налагается компилятором). Обычный способ решения этой проблемы состоит в том, чтобы осуществить этот вызов в выражении фильтра, как показано в следующем примере, в котором код исключения является аргументом функции фильтра, предоставляемой пользователем:

```
__except(MyFilter(GetExceptionCode())) {
}
```

В данном случае значение выражения фильтра, которое должно быть одним из трех указанных ранее значений, определяется и возвращается функцией фильтра. В свою очередь, для определения возвращаемого этой функцией значения используется код исключения; например, можно сделать так, чтобы фильтр передавал обработку исключений, возникающих при выполнении операций с плавающей точкой (FP-исключений, от *FloatingPoint* — плавающая точка), внешнему обработчику (возвращая значение EXCEPTION_CONTINUE_SEARCH), а обработку нарушений доступа к памяти — текущему обработчику (возвращая значение EXCEPTION_EXECUTE_HANDLER).

Число возможных кодов исключений, возвращаемых функцией GetExceptionCode, очень велико, однако их можно разделить на несколько категорий.

• Выполнение программой некорректных действий, например:

EXCEPTION_ACCESS_VIOLATION — попытка чтения или записи по адресу виртуальной памяти, к которой процесс не имеет доступа.

EXCEPTION_DATATYPE_MISALIGNMENT — многие процессоры, например, требуют чтобы данные типа DWORD выравнивались по четырехбайтовым границам.

EXCEPTION_NONCONTINUABLE_EXECUTION — значением выражения фильтра было EXCEPTION_CONTINUE_EXECUTION, но выполнения программы после возникновения исключения не может быть продолжено.

• Исключения, сгенерированные функциями распределения памяти HeapAlloc и HeapCreate, если они используют флаг HEAP_GENERATE_EXCEPTIONS (см. главу 5). Соответствующими значениями кода исключения являются STATUS_NO_MEMORY или EXCEPTION_ACCESS_VIOLATION.

• Коды определенных пользователем исключений, генерируемых путем вызова функции RaiseException, о чем говорится в подразделе "Исключения, генерируемые приложением".

• Коды различных арифметических исключений (особенно FP-исключений), например, EXCEPTION_INT_DIVIDE_BY_ZERO или EXCEPTION_FLT_OVERFLOW.

• Исключения, используемые отладчиками, например, EXCEPTION_BREAKPOINT или EXCEPTION_SINGLE_STEP.

Вам пригодится также функция GetExceptionInformation, которая может быть вызвана только из выражения фильтра и возвращает дополнительную информацию, включая информацию, специфическую для используемого процессора.

LPEXCEPTION_POINTERS GetExceptionINFORMATION(VOID)

Вся информация, как относящаяся, так и не относящаяся к процессору, содержится в структуре EXCEPTION_POINTERS, состоящей из двух других структур.

```
typedef struct _EXCEPTION_POINTERS {
 PEXCEPTION_RECORD ExceptionRecord;
 PCONTEXT ContextRecord;
} EXCEPTION POINTERS;
```

В структуру EXCEPTION_RECORD входит элемент ExceptionCode, набор возможных значений которого совпадает с набором значений, возвращаемых функцией GetExceptionCode. Элемент ExceptionFlags структуры EXCEPTION_RECORD может принимать значения 0 или EXCEPTION_NONCONTINUABLE, причем последнее значение указывает функции фильтра на то, что она не должна предпринимать попыток продолжения выполнения. К числу других элементов данных этой структуры относятся адрес виртуальной памяти ExceptionAddress и массив параметров ExceptionInformation. В случае исключения EXCEPTION_ACCESS_VIOLATION значение первого элемента этого массива указывает на то, какая именно из операций пыталась получить

доступ по недоступному адресу — записи (1) или чтения (0). Второй элемент содержит адрес виртуальный памяти.

Во втором элементе структуры EXCEPTION_POINTERS, а именно, элементе ContextRecord, содержится информация, относящаяся к процессору. Для каждого типа процессоров предусмотрены свои структуры, определения которых содержатся в файле <winnt.h>.

Резюме: последовательность обработки исключений

На рис. 4.2 в схематическом виде представлена последовательность событий, происходящих после возникновении исключения. Слева приведен программный код, а обведенные кружками цифры справа обозначают операции, выполняемые языковыми средствами поддержки времени выполнения. Отдельные элементы приведенной схемы имеют следующий смысл:

1. Возникло исключение; в данном случае это деление на ноль.
2. Управление передается обработчику исключений, в котором вычисляется выражение фильтра. Сначала вызывается функция GetExceptionCode, а затем ее возвращаемое значение используется в качестве аргумента функции Filter.
3. Функция фильтра выполняет действия, определяемые значением кода исключения.
4. В данном случае значением кода исключения является EXCEPTION_INT_DIVIDE_BY_ZERO.
5. Функция фильтра устанавливает, что должен быть выполнен код обработчика исключений, и поэтому возвращает значение EXCEPTION_EXECUTE_HANDLER.
6. Выполняется код обработчика исключений, связанного с оператором _except.
7. Управление передается за пределы блоков try и except.

```
_try {
        ...
        i = j / 0;
        ...
}
_except (Filter (GetExceptionCode ())) {
        ...
}
        ...
DWORD Filter (DWORD ExCode)
{
        switch (ExCode) {
        ...
        case EXCEPTION_INT_DIVIDE_BY_ZERO:
        ...
        return EXCEPTION_EXECUTE_HANDLER;
        case ...
        }
}
```

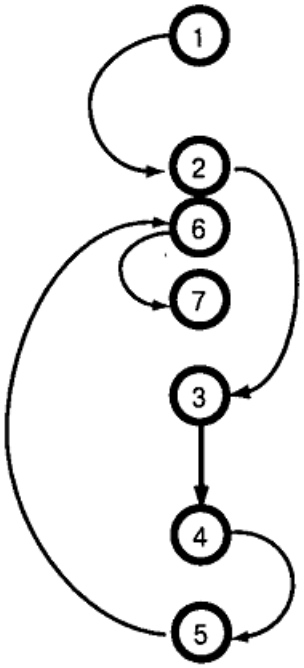

Рис. 4.2. Последовательность операций при обработке исключений

Исключения, возникающие при выполнении операций над числами с плавающей точкой

Существует семь различных кодов исключений, которые могут возникать при выполнении операций с использованием данных вещественного типа. Первоначально эти исключения отключены и не могут возникать до тех пор, пока с помощью функции _controlfp для них не будет предварительно задана специальная маска, не зависящая от типа процессора. Предусмотрены отдельные исключения для ситуаций антипереполнения, переполнения, деления на ноль, неточного результата и так далее, что иллюстрируется приведенным ниже фрагментом кода. Для активизации исключений определенного типа следует *отключить* соответствующий бит маски.

DWORD _controlfp(DWORD new, DWORD mask)

Фактическое значение маски определяется ее текущим значением (current_mask) и двумя аргументами следующим образом:

(current_mask & ~mask) | (new & mask)

Данная функция устанавливает лишь те из битов, указанных в аргументе new, которые разрешены аргументом mask. Биты, не

активизированные аргументом mask, не изменяются. Маска FP-исключений управляет также точностью, округлением и обработкой значений, соответствующих бесконечности, поэтому при активизации перечисленных исключений необходимо тщательно следить за тем, чтобы случайно не изменить эти установки.

Возвращаемым значением является фактическое значение маски. Так, при нулевых значениях обоих аргументов возвращаемым значением будет текущее значение маски (current_mask), что может быть использовано для восстановления маски, если впоследствии в этом возникнет необходимость. С другой стороны, если задать аргумент mask равным 0xFFFFFFFF, то регистр установится в new, что, например, может быть использовано для восстановления прежнего значения маски.

Обычно для того, чтобы разрешить исключения, связанные с выполнением операций над числами с плавающей точкой, в качестве аргумента mask используют константу MCW_EM, как продемонстрировано в следующем примере. Также заметьте, что при обработке FP-исключения оно должно быть сброшено путем использования функции _clearfp.

```
#include <float.h>
DWORD FPOld, FPNew; /* Старое и новое значения маски. */
…
FPOld = _controlfp(0, 0); /* Сохранить старую маску. */
/* Указать в качестве разрешенных шесть типов исключений. */
FPNew = FPOld & ~(EM_OVERFLOW | EM_UNDERFLOW | EM_INEXACT | EM_ZERODIVIDE | EM_DENORMAL | EM_INVALID);
/* Установить новую управляющую маску. Параметр MCW_EM объединяет шесть исключений, указанных в предыдущем операторе. */
_controlfp(FPNew, MCW_EM);
while(…) __try { /* Выполнить вычисления над числами с плавающей точкой. */
 … /* На этом участке кода может возникнуть FP-исключение. */
} __except(EXCEPTION_EXECUTE_HANDLER) {
 … /* Обработать FP-исключение. */
 _clearfp(); /* Сбросить исключение. */
 _controlfp(FPOld, 0xFFFFFFFF); /* Восстановить маску. */
}
```

В этом примере разрешены все возможные FP-исключения, кроме одного — EXCEPTION_FLT_STACK_CHECK, которое соответствует переполнению стека при выполнении операций над числами с плавающей точкой. Можно поступить и по-другому, разрешая отдельные исключения путем использования только выбранных масок исключений, например EM_OVERFLOW. Аналогичный код используется в программе 4.3 в контексте примера программного кода большего объема.

Ошибки и исключения

Под ошибками понимаются исключительные ситуации, которые время от времени могут возникать в известных местах программы. Так, обнаружение ошибок, возникающих во время выполнения системных вызовов, и немедленный вывод сообщений о них должны предусматриваться логикой работы самой программы. Поэтому программисты, как правило, явно включают в программный код участки, ответственные, например, за тестирование успешности завершения операции чтения данных из файла. В главе 2 для диагностики ошибок и принятия соответствующих мер была разработана функция ReportError.

С другой стороны, исключения могут возникать практически в любом месте программы, и поэтому организация явной проверки всех исключений невозможна или практически нецелесообразна. Примерами подобных ситуаций могут служить попытки деления на ноль или обращения к недоступным областям памяти.

Вместе с тем, указанные различия между ошибками и исключениями являются довольно условными. Windows позволяет управлять генерацией исключений, возникающих в случае нехватки памяти при ее распределении с использованием функций HeapAlloc и HeapCreate. Помимо этого, программы могут генерировать собственные исключения с кодами, определяемыми программистом, используя для этого функцию RaiseException, о чем далее будет говориться.

Обработчики исключений обеспечивают удобный механизм выхода из внутренних блоков или функций под управлением программы без использования операторов перехода goto или longjmp. Такая возможность оказывается особенно полезной, если блок получил доступ к таким, например, ресурсам, как открытые файлы, память или объекты синхронизации, поскольку обработчик может взять на себя задачу освобождения этих ресурсов. Возможно также продолжение работы программы после выполнения кода обработчика исключений, а не ее обязательное завершение. Кроме того, после выхода из блока программа может восстанавливать прежнее состояние системы, например маску FP-исключений. Именно в этом ключе обработчики используются во многих наших примерах.

Исключения, генерируемые приложением

Существует возможность формирования исключений в любой точке программы в процессе ее выполнения с помощью функции RaiseException. Это позволяет программе обнаруживать и обрабатывать возникающие ошибки как исключения.

VOID RaiseException(DWORD dwExceptionCode, DWORD dwExceptionFlags, DWORD cArguments, CONST DWORD *lpArguments)

Параметры:

dwExceptionCode — код исключения, определяемый пользователем. Бит 28 использовать нельзя, так как он зарезервирован системой. Для кода ошибки отводятся биты 27—0 (то есть все слово, кроме самого старшего шестнадцатеричного разряда). Бит 29 должен быть установлен, чтобы показать, что данное исключение имеет "пользовательскую" природу (а не относится к числу тех, которые предусмотрела Microsoft). В битах 31—30 содержится код серьезности ошибки, принимающий приведенные ниже значения, в которых результирующая старшая шестнадцатеричная цифра кода исключения представлена с установленным битом 29.

• 0 — успешное выполнение (старшая шестнадцатеричная цифра кода исключения равна 2).

• 1 — информационный код (старшая шестнадцатеричная цифра кода исключения равна 6).

• 2 — предупреждение (старшая шестнадцатеричная цифра кода исключения равна A).

• 3 — ошибка (старшая шестнадцатеричная цифра кода исключения равна E).

dwExceptionFlags — обычно устанавливается равным 0, тогда как установка значения EXCEPTION_NONCONTINUABLE будет указывать на то, что выражение фильтра не должно возвращать значение EXCEPTION_CONTINUE_EXECUTION; при попытке это сделать будет немедленно сгенерировано исключение EXCEPTION_NONCONTINUABLE_EXCEPTION.

lpArguments — этот указатель, если он не равен NULL, указывает на массив размера cArguments (третий параметр), содержащий 32-битовые значения, которые должны быть переданы выражению фильтра. Максимально возможное число этих значений ограничивается значением EXCEPTION_MAXIMUM_PARAMETERS, которое в настоящее время установлено равным 15. Для доступа к этой структуре следует использовать функцию GetExceptionInformation.

Заметьте, что невозможно сгенерировать исключение в другом процессе. В то же время, при весьма ограниченных условиях для этой цели могут быть использованы обработчики управляющих сигналов консоли, о чем говорится в конце этой главы и в главе 6.

Пример: обработка ошибок как исключений

В предыдущих примерах для обработки ошибок при выполнении системных вызовов и других ошибок используется функция ReportError. Эта функция прекращает выполнение процесса, если программист указал, что данная ошибка является критической. Вместе с тем, такой подход препятствует нормальному выходу из программы и не обеспечивает возможность продолжения работы программы после устранения последствий ошибки. Так, после отказа от задачи, которая привела к возникновению сбоя,

может потребоваться уничтожение временных файлов, созданных в процессе работы программы, или переход программы к выполнению других задач. Функции ReportError присущи и другие ограничения, перечень которых приводится ниже.

• Даже в тех случаях, когда было бы достаточно прекратить выполнения только одного потока, критическая ошибка приводит к остановке всего процесса (главу 7).

• Вместо завершения процесса может оказаться желательным продолжение выполнения программы.

• Во многих случаях становится невозможным освобождение ресурсов синхронизации (глава 8), например мьютексов.

При прекращении выполнения процесса (но не потоки) открытые дескрипторы будут закрываться, однако при этом необходимо учитывать другие отрицательные факторы.

Решение заключается в написании новой функции — ReportException. Если ошибка не является критической, эта функция вызывает функцию ReportError (разработанную в главе 2), которая выводит сообщение об ошибке. В случае же возникновения критической ошибки будет сгенерировано исключение. Система будет использовать обработчик исключений из вызывающего try-блока, и поэтому в действительности характер исключения может быть некритическим, если обработчик предоставляет программе возможность восстановиться после сбоя. По существу, функция ReportException дополняет обычные программные методы защиты от ошибок, ранее ограниченные функцией ReportError. В случае обнаружения ошибки обработчик позволяет программе продолжить свою работу после выполнения необходимых восстановительных действий. Эти возможности иллюстрирует программа 4.2.

Функция ReportException представлена в программе 4.1. Необходимые определения и заголовочные файлы не указаны, поскольку эта функция находится в том же модуле исходного кода, что и функция ReportError.

Программа 4.1. ReportException: функция вывода сообщений об исключениях

```
/* Расширение функции ReportError для генерации формируемого
приложением кода исключения вместо прекращения выполнения процесса. */
VOID ReportException(LPCTSTR UserMessage, DWORD
ExceptionCode)
/* Вывести сообщение о некритической ошибке. */
{
 ReportError(UserMessage, 0, TRUE);
 /* Если ошибка критическая, сгенерировать исключение. */
 if (ExceptionCode != 0) RaiseException((0x0FFFFFFF &
ExceptionCode) | 0xE0000000, 0, 0, NULL);
 return;
```

}

Функция ReportException используется в нескольких последующих примерах.

Модель сигналов, используемая в UNIX, значительно отличается от SEH. Сигналы могут быть пропущены или игнорированы, и логика их работы иная. Тем не менее, у этих моделей имеются и общие черты.

Значительная часть поддержки обработки сигналов в UNIX обеспечивается библиотекой C, ограниченная версия которой доступна также под управлением Windows. Во многих случаях в программах Windows вместо сигналов можно воспользоваться обработчиками управляющих сигналов консоли, описанными в конце данной главы.

Некоторые сигналы соответствуют исключениям Windows.

Перечень в некоторой мере ограниченных соответствий "сигнал-исключение" представлен ниже:

• SIGILL — EXCEPTION_PRIV_INSTRUCTION

• SIGSEGV — EXCEPTION_ACCESS_VIOLATION

• SIGFPE — семь различных исключений, связанных с выполнением операций над числами с плавающей точкой, например EXCEPTION_FLT_DIVIDE_BY_ZERO

• SIGUSR1 и SIGUSR2 — исключения, определяемые приложением

Функции RaiseException соответствует функция библиотеки C raise.

В Windows сигналы SIGILL, SIGSEGV и SIGFPE не генерируются, хотя функция raise может генерировать один из них. Сигнал SIGINT в Windows не поддерживается.

Функция UNIX kill (kill не входит в состав стандартной библиотеки C), которая посылает сигнал другому процессу, может быть сопоставлена функции Windows GenerateConsoleCtrlEvent (глава 6). Для ограниченного варианта SIGKILL в Windows имеются аналоги в виде функций TerminateProcess и TerminateThread, с помощью которых один процесс (или поток) может уничтожить другой, хотя при использовании этих функций необходимо соблюдать осторожность (см. главы 6 и 7).

Обработчики завершения

Обработчики завершения служат в основном тем же целям, что и обработчики исключений, но выполняются, когда поток покидает блок в результате нормального выполнения программы, а также когда возникает исключение. С другой стороны, обработчик завершения не может распознавать исключения.

Обработчик завершения строится с использования ключевого слова __finally в операторе try…finally. Структура этого оператора аналогична структуре оператора try…finally, но в ней отсутствует выражение фильтра. Как и обработчики исключений, обработчики завершения предоставляют удобные возможности для закрытия дескрипторов, освобождения ресурсов, восстановления масок и выполнения иных действий, направленных на

восстановление известного состояния системы после выхода из блока. Например, программа может выполнять операторы return внутри блока, оставляя всю работу по "уборке мусора" обработчику завершения. Благодаря этому отпадает необходимость во включении кода очистки в код самого блока или переходе к коду очистки при помощи оператора goto.

```
__try {
 /* Блок кода. */
} _finally  {
 /* Обработчик завершения (блок finally).  */
}
```

Выход из try-блока

Обработчик завершения выполняется всякий раз, когда в соответствии с логикой программы осуществляется выход из try-блока по одной из следующих причин:

• Достижение конца try-блока и "проваливание" в обработчик завершения.

• Выполнение одного из следующих операторов таким образом, что происходит выход за пределы блока:

return
break
goto[19]
longjmp
continue
__leave[20]

• Исключение.

Аварийное завершение

Любое завершение выполнения программы по причинам, отличным от достижения конца try-блока и "проваливания вниз" или выполнения оператора __leave, считается аварийным завершением. Результатом выполнения оператора __leave является переход в конец блока __try и передача управления вниз по тексту программы, что намного эффективнее простого использования оператора goto, поскольку не требует разворачивания стека. Для определения того, каким образом завершилось выполнение try-блока, в обработчике завершения используется следующая функция:

BOOL AbnormalTermination(VOID)

При аварийном завершении выполнения блока эта функция возвращает значение TRUE, при нормальном — FALSE.

Примечание

Завершение будет считаться аварийным, даже если, например, последним оператором try-блока был оператор return.

Выполнение обработчика завершения и выход из него

Обработчик завершения, или блок __finally, выполняется в контексте блока или функции, работу которых он отслеживает. Управление может переходить от оператора завершения к следующему оператору. Существует и другая возможность, когда обработчик завершения выполняет оператор передачи управления (return, break, continue, goto, longjmp или __leave). Еще одной возможностью является выход из обработчика по причине возникновения исключения.

Сочетание блоков finally и except

Один try-блок может иметь только один блок finally или только один блок except, но не может иметь оба указанных блока одновременно. Поэтому нижеприведенный код вызовет появление ошибок на стадии компиляции.

```
__try {
 /* Блок контролируемого кода. */
} __except (filter_expression) {
 /* Блок обработчика исключений. */
} __finally {
 /* Так делать нельзя! Это приведет к ошибке на стадии компиляции.
*/
}
```

Вместе с тем, допускается вложение одного блока в другой, что используется довольно часто. Нижеприведенный код является вполне работоспособным и обеспечивает гарантированное удаление временных файлов при выходе из цикла под управлением программы или в результате возникновения исключения. Эта методика оказывается удобной и в тех случаях, когда требуется обеспечить гарантированную отмену блокирования файлов, что будет использовано в программе 4.2. Кроме того, в коде имеется внутренний блок try…except, размещенный в том месте программы, где выполняются вычисления, в которых участвуют вещественные числа.

```
__try { /* Внешний блок try-except. */
 while (…) __try { /* Внутренний блок try-finally. */
  hFile = CreateFile(TempFile, …);
  if(…) __try { /* Внутренний блок try-except. */
   /* Разрешить FP-исключения. Выполнить вычисления. */
   …
  } __except(EXCEPTION_EXECUTE_HANDLER) {
   … /* Обработать FP-исключение. */
   _clearfp();
  }
  … /* Обработка исключений, не являющихся FP-исключениями. /*
 } __finally { /* Конец цикла while. */
  /* Выполняется на КАЖДОЙ итерации цикла. */
```

```
  CloseHandle(hFile);
  DeleteFile(TempFile);
 }
} __except (filter-expression) {
/* Обработчик исключений. */
}
```

Глобальное и локальное разворачивание стека

Исключения и аварийные завершения вызывают *глобальное разворачивание стека* (global stack unwind) в поиске обработчика, как было показано на рис. 4.1. Предположим, например, что в отслеживаемом блоке примера, приведенного в конце предыдущего раздела, исключение возникает прежде, чем активизируются FP-исключения. Тогда перед обработчиком исключения в стеке могут находиться многочисленные обработчики завершения.

Вспомните, что структура стека является динамической, как показано на рис. 4.1, и что в стеке, наряду с другими данными, хранятся данные обработчиков исключений и завершения. Фактическое содержимое стека в любой момент времени зависит от следующих факторов:

• *Статической* структуры программных блоков.

• *Динамической* структуры программы, отражаемой в последовательности открытых вызовов функций.

Обработчики завершения: завершение процессов и потоков

Обработчики завершения не выполняются, если выполнение процесса или потока было прекращено независимо от того, было ли это инициировано самим процессом путем использования функций ExitProcess или ExitThread, или вызвано извне, например, инициировано вызовом функций TerminateProcess или TerminateThread из другого места в программе. Поэтому ни одна из этих функций не должна вызываться процессом или потоком внутри блоков try…except или try…finally.

Обратите также внимание, что выполнение функции библиотеки C exit или возврат из функции main приводят к выходу из процесса.

SEH и обработка исключений в C++

При обработке исключений в C++ используются ключевые слова catch и throw, а сам механизм исключений реализован с использованием SEH. Тем не менее, обработка исключений в C++ и SEH — это разные вещи. Их совместное применение требует внимательного обращения, поскольку обработчики исключений, написанные пользователем и сгенерированные C++, могут взаимодействовать между собой и приводить к нежелательным последствиям. Например, находящийся в стеке обработчик __except может

перехватить исключение C++, в результате чего данное исключение так и не дойдет до обработчика C++.

Возможно и обратное, когда, например, обработчик C++ перехватит SEH-исключение, сгенерированное функцией RaiseException. Документация Microsoft рекомендует полностью отказаться от использования обработчиков Windows в программах на C++ и ограничиться применением в них только обработчиков исключений C++.

Кроме того, обработчики исключений или завершения Windows не осуществляют вызов деструкторов, что в ряде случаев необходимо для уничтожения экземпляров объектов C++.

Пример: использование обработчиков завершения для повышения качества программ

Обработчики исключений и завершения позволяют повысить надежность программ как за счет упрощения процедуры восстановления программы после возникновения ошибок и исключений, так и за счет гарантированного освобождения ресурсов и отмены блокирования файлов в критических ситуациях.

В программе toupper (программа 4.2) эти моменты иллюстрируются с привлечением идей, почерпнутых в программном коде предшествующих примеров. toupper обрабатывает несколько файлов, имена которых указываются в командной строке, переписывая их с преобразованием всех букв в верхний регистр. Имена преобразованных файлов получаются путем добавления префикса UC_ к исходным именам, и согласно "спецификации" программы запись поверх существующих файлов не производится. Преобразование файлов осуществляется в памяти машины, поэтому для каждого файла выделяется большая буферная область (достаточная для размещения всего файла). Кроме того, чтобы исключить любую возможность изменения файлов другими процессами, а также для того, чтобы вновь создаваемые выходные файлы строго соответствовали преобразованным входным файлам, оба вида файлов блокируются во время обработки. Понятно, что на каждой стадии обработки существует вероятность возникновения самых различных сбойных ситуаций, но в программе должна быть предусмотрена защита от подобных ошибок, и она должна располагать средствами, позволяющими ей восстановить свое нормальное состояние и попытаться обработать все остальные файлы, имена которых были указаны в командной строке. Программа 4.2 решает все эти задачи, обеспечивая разблокирование файлов во всех необходимых случаях без применения громоздкой логики операторов ветвления, к которым пришлось бы прибегнуть, если бы не были использованы средства SEH. Более подробные комментарии к программе содержатся в программном коде, находящемся на Web-сайте книги.

Программа 4.2. toupper: обработка файлов с восстановлением нормального состояния программы после сбоев

```
/* Глава 4. Команда toupper. */
/* Преобразование содержимое одного и более файлов с заменой всех букв на прописные. Имя выходного файла получается из имени входного файла добавлением к нему префикса UC_. */
#include "EvryThng.h"

int _tmain(DWORD argc, LPTSTR argv[]) {
 HANDLE hIn = INVALID_HANDLE_VALUE, hOut = INVALID_HANDLE_VALUE;
 DWORD FileSize, nXfer, iFile, j;
 CHAR OutFileName [256] = "", *pBuffer = NULL;
 OVERLAPPED ov = {0, 0, 0, 0, NULL}; /* Используется для блокирования файлов. */
 if (argc <= 1) ReportError(_T("Использование: toupper файлы"), 1, FALSE);
 /* Обработать все файлы, указанные в командной строке. */
 for (iFile = 1; iFile < argc; iFile++) __try { /* Блок исключений. */
  /* Все дескрипторы файлов недействительны, pBuffer == NULL, а файл OutFileName пуст. Выполнение этих условий обеспечивается обработчиками. */
  _stprintf(OutFileName, "UC_%s", argv[iFile]);
  __try { /* Внутренний блок try-finally. */
   /* Ошибка на любом шаге сгенерирует исключение, и следующий */
   /* файл будет обрабатываться только после "уборки мусора". */
   /* Объем работы по очистке зависит от того, в каком месте */
   /* программы возникла ошибка. */
   /* Создать выходной файл (завершается с ошибкой, если файл уже существует). */
    hIn = CreateFile(argv[iFile], GENERIC_READ, 0, NULL, OPEN_EXISTING, 0, NULL);
   if (hIn == INVALID_HANDLE_VALUE) ReportException(argv[iFile], 1);
  FileSize = GetFileSize(hIn, NULL);
    hOut = CreateFile(OutFileName, GENERIC_READ | GENERIC_WRITE, 0, NULL, CREATE_NEW, 0, NULL);
   if (hOut == INVALID_HANDLE_VALUE) ReportException(OutFileName, 1);
  /* Распределить память под содержимое файла. */
  pBuffer = malloc(FileSize);
   if (pBuffer == NULL) ReportException(_T("Ошибка при распределении памяти"), 1);
  /* Блокировать оба файла для обеспечения целостности копии. */
```

```
            if (!LockFileEx(hIn, LOCKFILE_FAIL_IMMEDIATELY, 0, FileSize,
0, &ov) ReportException(_T("Ошибка при блокировании входного файла"), 1);
            if (!LockFileEx(hOut, LOCKFILE_EXCLUSIVE_LOCK |
LOCKFILE_FAIL_IMMEDIATELY, 0, FileSize, 0, &ov)
ReportException(_T("Ошибка при блокировании выходного файла "), 1);
            /* Считать данные, преобразовать их и записать в выходной файл.
*/
            /* Освободить ресурсы при завершении обработки или
возникновении */
           /* ошибки; обработать следующий файл. */
            if (!ReadFile(hIn, pBuffer, FileSize, &nXfer, NULL))
ReportException(_T("Ошибка при чтении файла"), 1);
           for (j = 0; j < FileSize; j++) /* Преобразовать данные. */
            if (isalpha(pBuffer [j])) pBuffer[j] = toupper(pBuffer [j]);
            if(WriteFile(hOut, pBuffer, FileSize, &nXfer, NULL))
ReportException(T("Ошибка при записи в файл"), 1);
          } __finally {
           /*Освобождение блокировок, закрытие дескрипторов файлов,*/
           /*освобождение памяти и повторная инициализация */
           /*дескрипторов и указателя. */
           if (pBuffer != NULL) free (pBuffer);
           pBuffer = NULL;
           if (hIn != INVALID_HANDLE_VALUE) {
            UnlockFileEx(hIn, 0, FileSize, 0, &ov);
            CloseHandle(hIn);
            hIn = INVALID_HANDLE_VALUE;
           }
           if (hOut != INVALID_HANDLE_VALUE) {
            UnlockFileEx(hOut, 0, FileSize, 0, &ov);
            CloseHandle(hOut);
            hOut = INVALID_HANDLE_VALUE;
           }
           _tcscpy(OutFileName, _T(""));
          }
         }
         /* Конец основного цикла обработки файлов и блока try. */
         /* Обработчик исключений для тела цикла. */
         __except(EXCEPTION_EXECUTE_HANDLER) {
          _tprintf(_T("Ошибка при обработке файла %s\n"), argv[iFile]);
          DeleteFile (OutFileName);
         }
         _tprintf(_T("Обработаны все файлы, кроме указанных выше \n"));
         return 0;
        }
```

Пример: использование функции фильтра

Программа 4.3 представляет собой каркас программы, иллюстрирующей обработку исключений и завершения выполнения, в которой используется функция фильтра. Программа предлагает пользователю указать тип исключения, после чего продолжает работу для генерации исключения. Функция фильтра обрабатывает различные типы исключений по-разному; выбор вариантов, предусмотренных в программе, был совершенно произвольным и определялся исключительно целями демонстрации. В частности, программа обнаруживает попытки обращения к недоступным областям памяти, предоставляя адреса виртуальной памяти, по которым производилось такое обращение.

Блок __finally восстанавливает состояние маски FP-исключений. Совершенно очевидно, что восстановление состояния маски в данном случае, когда процесс уже должен завершаться, особого значения не имеет, но эта методика пригодится нам впоследствии, когда мы будем использовать ее на стадии завершения выполнения потока. Вообще говоря, процесс должен восстанавливать и системные ресурсы, например, удалять временные файлы, освобождать ресурсы синхронизации (глава 8) и отменять блокирование файлов (главы 3 и 6). Функция фильтра представлена в программе 4.4.

Данный пример не иллюстрирует обработку исключений, которые могут возникать при распределении памяти; эти исключения мы начнем интенсивно использовать в главе 5.

Программа 4.3. Exception: обработка исключений и завершения выполнения

```
#include "EvryThng.h"
#include <float.h>

DWORD Filter(LPEXCEPTION_POINTERS, LPDWORD);
double x = 1.0, y = 0.0;

int _tmain(int  argc, LPTSTR argv[]) {
 DWORD ECatgry, i = 0, ix,  iy = 0;
 LPDWORD pNull = NULL;
 BOOL Done = FALSE;
 DWORD FPOld, FPNew;
 FPOld = _controlfp(0,  0); /* Сохранить старую управляющую маску. */
 /* Разрешить FP-исключения. */
 FPNew = FPOld & ~(EM_OVERFLOW | EM_UNDERFLOW | EM_INEXACT | EM_ZERODIVIDE | EM_DENORMAL | EM_INVALID);
 _controlfp(FPNew, MCW_EM);
 while  (!Done) _try { /* Блок try-finally.  */
  _tprintf(_T("Введите тип исключения: "));
```

```
_tprintf(_T(" 1: Mem, 2: Int, 3: Flt 4: User 5: __leave "));
_tscanf(_T("%d"), &i);
__try { /* Блок try-except. */
 switch (i) {
 case 1: /* Исключение при обращении к памяти. */
  ix = *pNull;
  *pNull = 5;
  break;
  case 2: /* Исключение при выполнении арифметических операций над целыми числами. */
  ix = ix / iy;
  break;
 case 3: /* FP-исключение. */
  x = x / y;
  _tprintf(_T("x = %20e\n"), x);
  break;
 case 4: /* Пользовательское исключение. */
  ReportException(_T("Пользовательское исключение"), 1);
  break;
    case 5: /* Использовать оператор _leave для завершения выполнения.*/
  __leave;
 default:
  Done = TRUE;
 }
} /* Конец внутреннего блока __try. */
__except(Filter(GetExceptionInformation(), &ECatgry)) {
 switch(ECatgry) {
 case 0:
  _tprintf(_T("Неизвестное исключение\n"));
  break;
 case 1:
  _tprintf(_T("Исключение при обращении к памяти\n"));
  continue;
 case 2:
     _tprintf(_T("Исключение при выполнении арифметических операций над целыми числами \n"));
  break;
 case 3:
  _tprintf(_T("FP-исключение\n"));
  _clearfp();
  break;
 case 10:
  _tprintf(_T("Пользовательское исключение\n"));
  break;
```

```
 default:
  _tprintf(_T("Неизвестное исключение\n"));
  break;
 } /* Конец оператора switch. */
 _tprintf(_T("Конец обработчика\n"));
}
/* Конец блока try-except. */
} /* Конец цикла while – ниже находится обработчик завершения. */
__finally { /* Это часть цикла while. */
_tprintf(_T("Аварийное завершение?: %d\n"),
AbnormalTermination());
}
_controlfp(FPOld, 0xFFFFFFFF); /* Восстановить старую FP-маску.*/
return 0;
}
```

Программа 4.4 представляет функцию фильтра, используемую в программе 4.3. Эта функция просто проверяет и классифицирует различные возможные значения кодов исключений. В программном коде, размещенном на Web-сайте книги, проверяется каждое из возможных значений, в то время как приведенная ниже функция осуществляет проверку лишь тех из них, которые нужны для тестовой программы.

Программа 4.4. Функция Filter

```
static DWORD Filter(LPEXCEPTION_POINTERS pExP, LPDWORD
ECatgry)
/* Классификация исключений и выбор соответствующего действия.
*/
{
DWORD ExCode, ReadWrite, VirtAddr;
ExCode = pExP->ExceptionRecord->ExceptionCode;
_tprintf(_T("Filter. ExCode:. %x\n"), ExCode);
if ((0x20000000 & ExCode) != 0) { /* Пользовательское исключение.
*/
 *ECatgry =10;
 return EXCEPTION_EXECUTE_HANDLER;
}
switch (ExCode) {
case EXCEPTION_ACCESS_VIOLATION:
 ReadWrite = /* Операция чтения или записи? */
  pExP->ExceptionRecord->ExceptionInformation[0];
 VirtAddr = /* Адрес сбоя в виртуальный памяти. */
  pExP->ExceptionRecord->ExceptionInformation [1];
  _tprintf(_T("Нарушение доступа. Чтение/запись: %d. Адрес: %x\n"),
ReadWrite, VirtAddr);
 *ECatgry = 1;
```

```
 return EXCEPTION_EXECUTE_HANDLER;
case EXCEPTION_INT_DIVIDE_BY_ZERO:
case EXCEPTION_INT_OVERFLOW:
 *ECatgry = 2;
 return EXCEPTION_EXECUTE_HANDLER;
case EXCEPTION_FLT_DIVIDE_BY_ZERO:
case EXCEPTION_FLT_OVERFLOW:
 _tprintf(_T("FP-исключение — слишком большое значение.\n"));
 *ECatgry = 3;
 _clearfp();
 return (DWORD)EXCEPTION_EXECUTE_HANDLER;
default:
 *ECatgry = 0;
 return EXCEPTION_CONTINUE_SEARCH;
 }
}
```

Обработчики управляющих сигналов консоли

Обработчики исключений могут реагировать на самые разнообразные события, но они не в состоянии обнаруживать такие ситуации, как выход пользователя из системы или нажатие комбинации клавиш <Ctrl+C> на клавиатуре с целью прекращения выполнения программы. Для обработки таких событий требуются обработчики управляющих сигналов консоли.

Функция SetConsoleCtrlHandler позволяет одной или нескольким указанным функциям выполняться в ответ на получение сигналов Ctrl-c, Ctrl-break или одного из трех других сигналов, связанных с консолью. Функция GenerateConsoleCtrlEvent, описанная в главе 6, также генерирует эти сигналы, а, кроме того, все эти сигналы могут посылаться другим процессам, совместно использующим ту же консоль. Обработчиками сигналов являются указанные пользователем функции, которые возвращают булевские значения и принимают единственный аргумент типа DWORD, идентифицирующий фактический сигнал.

С одним сигналом могут быть ассоциированы несколько обработчиков, причем обработчики можно добавлять и удалять. Функция, которая используется для добавления и удаления обработчиков, имеет следующий вид:

BOOL SetConsoleCtrlHandler(PHANDLER_ROUTINE HandlerRoutine, BOOL Add)

Значению флага Add, равному TRUE, соответствует добавление процедуры обработчика, в противном случае происходит удаление процедуры из списка процедур обработки управляющих сигналов консоли. Заметьте, что тип сигнала при вызове функции не конкретизируется.

Тестирование с целью проверки того, какой именно сигнал получен, должен выполнять сам обработчик.

Процедура обработчика возвращает булевское значение и принимает единственный параметр типа DWORD, идентифицирующий фактический сигнал. Использованное в объявлении имя обработчика (HandlerRoutine) является заменителем, и программист может выбирать его по своему усмотрению.

Ниже приводятся дополнительные полезные сведения, касающиеся использования обработчиков управляющих сигналов консоли.

• Если значение параметра HandlerRoutine равно NULL, а параметра Add — TRUE, то сигналы Ctrl-c будут игнорироваться.

• Если при вызове функции SetConsoleMode был задан параметр ENABLE_PROCESSED_INPUT то комбинация <Ctrl+C> будет обрабатываться не как сигнал, а как клавиатурный ввод.

• Процедура обработчика фактически выполняется как независимый поток внутри процесса. При этом выполнение основной программы, как показано в следующем примере, не приостанавливается.

• Формирование исключения в обработчике *не вызовет* исключения в потоке, выполнение которого было прервано, поскольку исключения применяются только к потокам, а не к процессу в целом. Если вы хотите организовать связь с прерванным потоком, используйте переменную, как в следующем примере, или метод синхронизации (глава 8).

Между исключениями и сигналами существует важное отличие. Сигналы применяются к процессу в целом, тогда как исключения — только к потоку, выполняющему код, в котором возникло исключение.

BOOL *HandlerRoutine*(DWORD dwCtrlType)

dwCtrlType идентифицирует фактический сигнал (или *событие)* и может принимать одно из следующих пяти значений:

1. CTRL_C_EVENT указывает на то, что комбинация <Ctrl+C> должна восприниматься как клавиатурный ввод.
2. CTRL_CLOSE_EVENT указывает на закрытие окна консоли.
3. CTRL_BREAK_EVENT указывает на сигнал Ctrl-break.
4. CTRL_LOGOFF_EVENT указывает на выход пользователя из системы.
5. CTRL_SHUTDOWN_EVENT указывает на завершение работы системы.

Обработчик сигналов может выполнять операции по "уборке мусора" точно так же, как это делают обработчики исключений и завершения. В случае успешной обработки сигнала обработчик должен вернуть значение TRUE. Если обработчик возвращает значение FALSE, выполняется следующая функция обработчика из числа тех, что указаны в списке. Обработчики сигналов выполняются в порядке, обратном порядку их установки, так что первым будет выполняться самый последний из установленных обработчиков, а системный обработчик будет выполняться самым последним.

Пример: обработчик управляющих сигналов консоли

В программе 4.5 организован бесконечный цикл, в котором каждые 5 секунд вызывается функция Beep, подающая звуковой сигнал. Пользователь может завершить выполнение программы, нажав комбинацию клавиш <Ctrl+C> или закрыв консоль. Процедура обработчика выводит на экран сообщение, выжидает 10 секунд, после чего, казалось бы, выполнение программы должно завершиться с возвратом значения TRUE. Однако в действительности основная программа обнаруживает флаг Exit и останавливает процесс. Это демонстрирует параллельную природу выполнения процедуры обработчика; заметьте, что объем выходной информации обработчика сигналов зависит от временных характеристик сигнала. Обработчики управляющих сигналов консоли будут использоваться также в примерах, приводимых в следующих главах.

Обратите внимание на использование макроса WINAPI; он применяется к пользовательским функциям, передаваемым в качестве аргументов функциям Windows, чтобы гарантировать выполнение соответствующих соглашений о вызовах. Этот макрос определен в заголовочном файле Microsoft C WTYPES.H.

Программа 4.5. Ctrlc: программа обработки сигналов

```
/* Глава 4. Ctrlc.c */
/* Перехватчик событий консоли. */
#include "EvryThng.h"

static BOOL WINAPI Handler(DWORD CtrlEvent); /* См. WTYPES.H. */
volatile static BOOL Exit = FALSE;

int _tmain(int argc, LPTSTR argv[])
 /* Периодическая подача звукового сигнала до поступления сигнала о прекращении выполнения. */
{
 /* Добавить обработчик событий. */
 if (!SetConsoleCtrlHandler(Handler, TRUE)) ReportError(_T("Ошибка при установке обработчика событий."), 1, TRUE);
 while (!Exit) {
  Sleep (5000); /* Подача звукового сигнала каждые 5 секунд. */
  Beep(1000 /* Частота. */, 250 /* Длительность. */);
 }
 _tprintf(_T("Прекращение выполнения программы по требованию.\n"));
 return 0;
}
```

```
BOOL WINAPI Handler (DWORD CtrlEvent) {
 Exit = TRUE;
 switch (CntrlEvent) {
 /* Увидите ли вы второе сообщения обработчика, зависит от соотношения временных параметров. */
 case CTRL_C_EVENT:
  _tprintf(_T("Получен сигнал Ctrl-c. Выход из обработчика через 10 секунд.\n"));
  Sleep(4000); /* Уменьшите это значение, чтобы получить другой эффект. */
 _tprintf(_T("Выход из обработчика через 6 секунд.\n"));
 Sleep(6000); /* Попробуйте уменьшить и это значение. */
 return TRUE; /* TRUE указывает на успешную обработку сигнала. */
 case CTRL_CLOSE_EVENT:
 _tprintf(_T("Выход из обработчика через 10 секунд.\n"));
 Sleep(4000);
 _tprintf(_T ("Выход из обработчика через 6 секунд.\n"));
 Sleep (6000); /* Попробуйте уменьшить и это значение. */
  return TRUE; /* Попробуйте возвратить FALSE. Приводит ли это к изменению поведения программы? */
 default:
   _tprintf(_T("Событие: %d. Выход из обработчика через 10 секунд.\n"), CntrlEvent);
 Sleep(4000);
 _tprintf(_T("Выход из обработчика через 6 секунд.\n"));
 Sleep(6000);
 return TRUE;
 }
}
```

Векторная обработка исключений

Функции обработки исключений можно непосредственно связывать с исключениями, точно так же, как обработчики управляющих сигналов консоли можно связывать с управляющими событиями консоли. В этом случае, если возникает исключение, то первыми, еще до того, как система начнет разворачивать стек в поиске структурных обработчиков исключений, будут вызываться *векторные обработчики исключений* (vectored exception handlers). При этом никакие ключевые слова, аналогичные __try или __catch, не требуются. Такая возможность предоставляется только в Windows XP и Windows Server 2003.

Работа с векторными обработчиками исключений (Vectored Exception Handling, VEH) напоминает работу с обработчиками управляющих сигналов консоли, хотя детали и отличаются. Для добавления, или регистрации, обработчика служит функция AddVectoredExceptionHandler.

PVOID AddVectoredExceptionHandler(ULONG FirstHandler, PVECTORED_EXCEPTION_HANDLER VectoredHandler)

Обработчики можно связывать в цепочки, поэтому первый параметр **First-Handler** указывает, что при возникновении исключения обработчик должен вызываться либо первым (ненулевое значение), либо последним (нулевое значение). Последующие вызовы функции AddVectoredExceptionHandler могут изменить этот порядок. Например, если добавляются два обработчика, причем для каждого из них задаются нулевые значения параметра FirstHandler, то они будут вызываться в том порядке, в котором добавлялись.

Функция RemoveVectoredExceptionHandler, прекращающая регистрацию векторного обработчика исключений, требует задания единственного параметра, адреса обработчика, и в случае успешного выполнения возвращает ненулевое значение.

Функция AddVectoredExceptionHandler в случае успешного выполнения возвращает адрес обработчика исключений, т.е. VectoredHandler. Возвращаемое значение NULL указывает на неудачное завершение выполнения функции.

VectorHandler — это указатель на функцию обработчика, которая имеет следующий прототип:

LONG WINAPI VectoredHandler(PEXCEPTION_POINTERS ExceptionInfo)

PEXCEPTION_POINTERS — адрес структуры EXCEPTION_POINTERS, которая содержит как информацию, зависящую от типа процессора, так и информацию общего характера. Это та же структура, которую возвращает функция GetExceptionInformation и которая уже использовалась нами в программе 4.4.

От функции VEH-обработчика требуется, чтобы она выполнялась быстро и никогда не получала доступа к объектам синхронизации, таким как мьютекс (см. главу 8). В большинстве случаев VEH-обработчики просто обращаются к структуре исключения, выполняют некоторую минимальную обработку (например, устанавливают флаг) и осуществляют возврат. Возможны два возвращаемых значения, с которыми мы уже встречались при обсуждении SEH-обработчиков.

1. EXCEPTION_CONTINUE_EXECUTION — обработчики далее не выполняются, обработка средствами SEH не производится, и управление передается в ту точку программы, в которой возникло исключение. Как и в случае SEH, это оказывается возможным не всегда.

2. EXCEPTION_CONTINUE_SEARCH — выполняется следующий VEH-обработчик, если таковой имеется. Если обработчиков больше нет, разворачивается стек для поиска SEH-обработчиков.

В упражнении 4.9 вам предлагается добавить VEH в программы 4.3 и 4.4.

Резюме

Структурная обработка исключений в Windows предоставляет в распоряжение разработчиков механизм повышения надежности, благодаря которому С-программы могут адекватно реагировать на ошибки и исключения и восстанавливаться после возникновения сбоев в процессе выполнения. Методы обработки исключений отличаются высокой эффективностью, и их применение делает структуру программ более понятной, что облегчает их сопровождение и улучшает их качественные характеристики. В большинстве других языков и ОС также реализованы аналогичные подходы, однако решение Windows обеспечивает возможность точного анализа природы возникающих исключений.

Обработчики управляющих сигналов консоли позволяют реагировать на внешние события, наступление которых не сопровождается генерацией исключений. Векторная обработка исключений является новейшим средством, обеспечивающим выполнение соответствующих функций еще до того, как начнется выполнение SEH-процедур. Механизм VEH аналогичен обычному механизму векторных прерываний.

Упражнения

4.1. Расширьте возможности программы 4.2 путем предоставления при каждом вызове функции ReportException достаточно большого объема информации, чтобы обработчик исключений в своих сообщениях указывал точную природу возникающих ошибок и удалял выходные файлы, если их содержимое оказывается незначащим.

4.2. Расширьте возможности программы 4.3 за счет генерации таких исключений, связанных с нарушениями доступа к памяти, как выход индекса за пределы допустимого диапазона, а также исключений, обусловленных сбоями при выполнении арифметических операций, и других FP-исключений, не предусмотренных в программе 4.3.

4.3. Дополните программу 4.3 таким образом, чтобы она выводила на печать фактическое значение FP-маски после разрешения исключений. Все ли исключения оказались действительно разрешенными? Объясните результаты.

4.4. Какие значения вы в действительности получаете после возникновения таких FP-исключений, как деление на ноль? Можете ли вы установить результат в функции фильтра, как это пытается делать программа 4.3?

4.5. Что произойдет при выполнении программы 4.3, если не сбросить FP-исключение? Объясните результат. *Подсказка.* Запросите дополнительное исключение после возникновения FP-исключения.

4.6. Расширьте возможности программы 4.5 таким образом, чтобы процедура обработчика формировала исключение, а не возврат из функции. Объясните полученные результаты.

4.7. Расширьте возможности программы 4.5 таким образом, чтобы она могла обрабатывать сигналы, указывающие на выход пользователя из системы и завершение работы системы.

4.8. Экспериментальным путем убедитесь в том, что процедура обработчика в программе 4.5 выполняется параллельно с основной программой.

4.9. Усовершенствуйте программы 4.3 и 4.4. В частности, организуйте обработку арифметических и FP-исключений до активизации SEH.