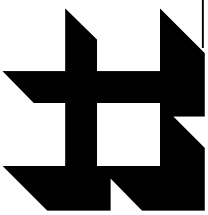

ГЛАВА 14

Обработка ошибок

Как говорит народная мудрость — не ошибается только тот, кто ничего не делает. Наши программы должны выполнять заданные нами функции, стало быть, при их работе могут возникнуть ошибочные ситуации. О том, какие бывают ошибки и как их обрабатывать, и пойдет речь в этой главе.

Ошибки в программах можно подразделить на:

- синтаксические;
- ошибки времени выполнения (Run-Time Error);
- ошибки в логике программы.

Синтаксические ошибки

Синтаксическая ошибка — это нарушение правил построения команды, функции, имени переменной, наименования объекта, наиболее вероятной причиной синтаксической ошибки может быть неверный ввод. Часто это бывает несоответствие количества открытых и закрытых скобок, наличие лишних или отсутствие необходимых запятых, описки в ключевых словах или их пропуски, несоответствие типов данных или необъявленные переменные.

Ошибки в логике программы

При наличии ошибок в логике программа может выполняться, но работает неправильно, выдавая непредсказуемые или заведомо неверные результаты. Логические ошибки — наиболее сложные, их труднее всего выявить. Вероятной причиной ошибки в логике программы может быть неверный алгоритм построения программы, некорректное использование управляющих структур и т. д. В качестве примера такой ошибки можно привести следующую конструкцию:

```
x=25
```

```
DO CASE
        CASE x>15
                <Команды>
        CASE x>20
           <Команды>
ENDCASE
```

Ошибки времени выполнения (Run-Time Error)

Ошибки времени выполнения (обычно такие ошибки называют *исключениями*) происходят после того, как приложение начинает выполняться. Например, ошибки могут возникать во время выполнения следующих действий:

- запись в несуществующий файл;
- попытка открыть открытую таблицу или попытка выбрать закрытую таблицу;
- конфликты данных;
- деление на нуль и другие.

Инструменты для определения ошибок

Синтаксическая ошибка может быть обнаружена синтаксическим анализатором уже при вводе кода программы в окне редактора или командном окне. В этом случае неверно написанная строка будет подчеркнута.

При попытке откомпилировать и запустить программу, содержащую синтаксическую ошибку, компиляция прекращается и на экран выводится окно редактирования. Ошибочная строка при этом подсвечивается. На экран также выводится диалоговое окно, содержащее текст сообщения об ошибке и кнопки: **Cancel** — прекратить выполнение программы, **Suspend** — приостановить выполнение, **Ignore** — игнорировать, **Help** — вывести на экран файл помощи Visual FoxPro с более подробным описанием ошибки.

Если произошла ошибка времени выполнения, для ее локализации и устранения можно применить ряд команд Visual FoxPro, специально предназначенных для этого:

- `ON ERROR` — глобальный обработчик ошибок;
- `TRY...CATCH...FINALLY` — обработчик локальных ошибок в программном коде;
- событие `Error` объектов — обработчик локальных ошибок, связанных с событиями объектов.

Для локализации и устранения логических ошибок необходимо отслеживать ход выполнения программы. В этом случае не обойтись без отладчика (Debugger), **Coverage Profiler** и тщательной проверки алгоритма работы. **Debugger** поможет следить за последовательностью действий и изменением значений. **Coverage Profiler** необходим

для того, чтобы проконтролировать, как фактически используется код: какие фрагменты кода насколько часто используются.

При возникновении ошибки в методе или событии объекта можно (и нужно!) использовать обработчик события `Error` объекта.

> ***Примечание***
> Если ваше приложение не содержит процедуры обработки ошибок, то используется встроенный обработчик ошибок, который выводит диалоговое окно **Program Error**.

Обработка ошибочных ситуаций

Пользователи, способные внятно рассказать разработчику, что же все-таки случилось с программой — большая редкость. Поэтому многие разработчики записывают ошибки в log-файл. Это может быть текстовый файл или таблица, куда записывается информация о времени ошибки, имени пользователя, номере ошибки, номере строки кода, имени процедуры, в которой произошла ошибка.

Важно также проинформировать пользователя о произошедшей ошибке. Информация, выводимая на экран, должна быть ясной и понятной пользователю и предлагать выход из ошибочной ситуации. Например, в случае ошибки печати можно выдать на экран сообщение, предлагающее проверить, включен ли принтер, есть ли в нем бумага.

В случае ошибки пользователю обязательно должен быть предложен выход: попробовать выполнить операцию позже, попытаться повторно выполнить команду либо отменить свои действия.

А теперь — самое важное: нужно попытаться "поймать ошибку", для этого, возможно, придется использовать многое из инструментария определения ошибок. Набор вашего "вооружения" достаточно широк — от стандартного обработчика ошибок до **Coverage Profiler**.

Стандартный обработчик ошибок

Стандартный обработчик ошибок встроен в Visual FoxPro. При возникновении ошибки происходит прерывание текущей процедуры и вызывается процедура обработки исключения.

Команда *ON ERROR*

Команда `ON ERROR` позволяет вместо стандартного обработчика ошибок VFP вызывать пользовательскую процедуру обработки ошибки (листинг 14.1). Синтаксис команды:

```
ON ERROR [команда],
```

где [*команда*] — это процедура или функция, которая будет вызываться при возникновении ошибки, например

```
ON ERROR ErrorHandler(ERROR(),PROGRAM(),LINENO())
```

```
FUNCTION ErrorHandler(nError,cMethod,nLine)
LOCAL lcErrorMsg,lcCodeLineMsg
WAIT CLEAR
lcErrorMsg=MESSAGE()+CHR(13)+CHR(13)
lcErrorMsg=lcErrorMsg+"Method:    "+cMethod
lcCodeLineMsg=MESSAGE(1)
IF BETWEEN(nLine,1,10000) AND NOT lcCodeLineMsg="..."
        lcErrorMsg=lcErrorMsg+CHR(13)+"Line:         "+ALLTRIM(STR(nLine))
        IF NOT EMPTY(lcCodeLineMsg)
        lcErrorMsg=lcErrorMsg+CHR(13)+CHR(13)+lcCodeLineMsg
        ENDIF
ENDIF
IF MESSAGEBOX(lcErrorMsg,17,_screen.Caption)#1
        ON ERROR
        RETURN .F.
ENDIF
ENDFUNC
```

Параметры, которые можно передать обработчику ошибок для получения наиболее полной информации об ошибке (табл. 14.1).

Таблица 14.1. *Функции для уточнения информации об ошибке*

Функция	Описание
AERROR()	Помещает информацию о наиболее поздней ошибке в массив
ERROR()	Возвращает номер ошибки
LINENO([1])	Возвращает (без параметра) номер выполняемой строки программы относительно первой строки основной программы. С параметром возвращает номер строки относительно первой строки компонента программы
MESSAGE([1])	Возвращает текущее сообщение об ошибке в виде символьной строки или (с параметром) содержимое строки программы, вызвавшей ошибку
PROGRAM(nLevel)	Без параметра возвращает имя выполняемой программы, в которой произошла ошибка. nLevel показывает количество отображаемых уровней (не более 128)
SYS(16, nLevel)	Возвращает имя файла с указанием пути, в котором находится выполняемая процедура. Имя файла не будет возвращено, если выполняемая программа является частью приложения
SYS(2018)	Включает дополнительные сведения об ошибке в строку сообщения об ошибке

После выполнения команды программа продолжает выполняться со строки, следующей за ON ERROR. Если процедура обработки ошибок включает команду RETRY, то ошибочная строка будет выполнена повторно. Для восстановления стандартного обработчика ошибок достаточно использовать команду ON ERROR без параметра.

Конструкция *TRY... CATCH... FINALLY*

Эта управляющая конструкция применяется для обработки ошибок и исключений, которые могут произойти во время выполнения. Синтаксис см. в приложении 2, № 264.

TRY...CATCH...FINALLY содержат блоки кодов для деклараций TRY, CATCH или FINALLY. В этих блоках можно выполнять следующие задачи по обработке ошибок:

- генерировать исключение;
- захватывать исключения;
- использовать вложения TRY...CATCH...FINALLY блоков;
- передавать исключения;
- немедленно выходить из TRY...CATCH...FINALLY;
- использовать команды;
- запускать завершающие декларации.

Принцип работы конструкции следующий: выполняются команды блока TRY, в этом блоке вы можете указывать наборы команд, которые могут привести к появлению ошибки в режиме выполнения. Если при выполнении ошибок не произошло, блок CATCH пропускается, управление передается блоку FINALLY, если он, конечно, имеется, а если он не определен, то исполняется первая строка, которая следует за декларацией ENDTRY.

Генерация исключения

Если при выполнении блока TRY происходит ошибка, Visual FoxPro генерирует исключение, которое представляет собой условие, которое требует обработки ошибки в отдельной процедуре. Программа после обнаружения ошибки "посылает" исключение, и подходящая декларация в блоке CATCH обрабатывает исключение. После обработки ошибки управление передается блоку FINALLY.

Захват исключения

Когда происходит ошибка в блоке TRY, управление переходит первой декларации CATCH. Можно определить сколько угодно деклараций CATCH, а можно и вовсе их не определять. Программа проверяет декларации CATCH в том порядке, в котором они расположены в структуре, и пытается определить, имеется ли процедура, способная

обработать эту ошибку. Если программа находит такую декларацию, то выполняется код, заключенный между искомой и следующей декларацией.

Декларация CATCH может содержать дополнительные опции TO и WHEN. При определении опции TO используется переменная памяти VarName, которая хранит ссылку на объект Exception, который генерируется только тогда, когда генерируется исключение. Если для выполнения декларации CATCH необходимо установить некоторое условие, то оно указывается в опции WHEN и должно удовлетворять критерию "истинно" (.T.). Если опции TO и WHEN отсутствуют, то декларация CATCH оценивается как CATCH WHEN .T..

После того как будет выполнен блок кода в CATCH, программа уже не возвращается в TRY и не выполняет другие декларации CATCH. Управление передается FINALLY, а если ее нет, то строке, следующей за ENDTRY. В конструкции TRY...CATCH...FINALLY могут отсутствовать как CATCH, так и FINALLY. В таком случае Visual FoxPro посылает сгенерированную ошибку в обработчик ошибок, имеющий более высокий уровень.

Приоритеты обработчиков ошибок

Если ошибка произошла в методе объекта, который вызывался из блока TRY, Visual FoxPro ищет процедуру обработки ошибок этого объекта. Например, ошибка генерируется в строке MyForm.SetAll(), которая находится в блоке TRY. Если существует код для обработки события Error, тогда событие Error имеет преимущество. Если в событии Error код обработки не предусмотрен, то производится попытка обработать ошибку в декларации CATCH.

Декларации TRY...CATCH должны вызывать только методы, имеющие событие Error.

Приоритет обработчиков ошибок для кода объекта:

- TRY...CATCH в методе, где произошло исключение;
- событие Error, если оно определено;
- TRY...CATCH, который выше в цепочке вызовов и который может существовать в методе верхнего уровня;
- системный обработчик ошибок Visual FoxPro.

Вид обработчика, принявшего ошибку, можно определить, используя функцию SYS(2410). Функция SYS(2410) возвращает тип используемого обработчика (табл. 14.2).

Таблица 14.2. *Определение типа обработчика с помощью функции SYS(2410)*

Возвращаемое значение (Character)	Описание обработчика
0	Стандартный
1	TRY... CATCH... FINALLY
2	Обработчик события Error

3	ON ERROR

Функция может вернуть неверное значение, если оператор CATH в конструкции TRY...CATCH...FINALLY отсутствует или его опция WHEN <выражение> вычислена как .F. (листинг 14.2).

```
TRY
   set tableprompt off
   select * from newtable
** Если таблицы newtable нет, то ошибка обрабатывается Catch-блоком
CATCH
 =messagebox("Ошибка! Таблица NewTable  отсутствует.")
ENDTRY
```

Приведем еще один пример для понимания работы конструкции (листинг 14.3).

```
TRY
   USE Table1 in 0
   USE Table2 in 0
*Здесь обработка данных
CATCH
*Здесь обработка ошибок
FINALLY
* Сюда попадаем и при успешном выполнении кода, и при неудачном открытии какой-
* либо таблицы
   USE in Table1
   USE in Table2
ENDTRY
```

Вложения блоков *TRY...CATCH*

Можно осуществлять вложения блоков TRY...CATCH. Это значит, что вы можете вставлять структуру TRY...CATCH...FINALLY внутри блоков деклараций TRY, CATCH или FINALLY.

Передача исключений

Если есть необходимость "перенаправить" или *передать исключение более высокому по уровню обработчику*, можно использовать декларацию THROW. THROW может быть вызван из любого блока кода структуры TRY...CATCH...FINALLY. Присутствие THROW в

блоке прерывает выполнение текущего блока. Однако использовать THROW можно только в случае, если существует обработчик для захвата исключения. В противном случае произойдет выход из программы. THROW не может быть вызван из командного окна.

Объекты исключений

Когда в блоке TRY происходит ошибка, Visual FoxPro автоматически создает объект Exception, который содержит детали случившейся ошибки. Декларация CATCH обрабатывает те ошибки, на которые устанавливается ссылка в переменной памяти VarName опции TO. Объект Exception обладает рядом свойств и методов, которые приведены в табл. 14.3.

Таблица 14.3. *Свойства, методы и события объекта* Exception

Свойства объекта Exception	**Описание**
BaseClass	Содержит имя базового класса Visual FoxPro, на котором основан объект. Информация только для чтения
Class	Содержит имя класса, на котором основан объект. Информация только для чтения
ClassLibrary	Содержит имя библиотеки классов, определенной пользователем. Информация только для чтения
Comment	Информация о загрузке объекта. Доступно во время проектирования и во время выполнения
Details	Содержит дополнительную информацию об объекте, которая определяется свойством Message. Читается и записывается во время выполнения
ErrorNo	Содержит номер ошибки объекта Exception. Доступно для записи/чтения в режиме выполнения
LineContents	Определяет содержимое строки, которая породила объект Exception. Доступно для записи/чтения в режиме выполнения
LineNo	Определяет номер строки, которая породила объект Exception. Доступно для записи/чтения в режиме выполнения
Message	Определяет фактическое сообщение об ошибке
Name	Содержит имя, используемое для того, чтобы ссылаться на него в коде. Доступно во время проектирования и запись/чтение в режиме run time
Parent	Ссылка на объект-контейнер. Только для чтения во время выполнения
ParentClass	Возвращает имя базового класса объекта. Только для чтения

Procedure	Определяет имя процедуры или метода, в которых происходит исключение. Доступно для записи/чтения в режиме run time
StackLevel	Содержит уровень стека для программы, в которой происходит исключение. Доступны чтение/запись в режиме run time
Tag	Загружает любые дополнительные данные, необходимые для вашей программы. Доступно в проектном режиме и режиме run time
UserValue	Определяет значение, переданное утверждением THROW при генерации исключения. Вы можете использовать эту величину как объектную ссылку. Доступны чтение/запись во времени выполнения

Таблица 14.3 *(окончание)*

Методы объекта Exception	**Описание**
AddProperty	Добавляет новое свойство для объекта
ReadExpression	Возвращает значение свойства в окне свойств
ReadMethod	Возвращает текст определенного метода
ResetToDefault	Восстанавливает свойство, событие или метод в состояние "по умолчанию"
SaveAsClass	Сохраняет пример объекта как определение класса в библиотеке класса
WriteExpression	Записывает выражение в свойство
WriteMethod	Записывает определенный текст в метод
События объекта Exception	
Destroy	Происходит при освобождении объекта
Error	Происходит при ошибке во время выполнения
Init	Происходит во время создания объекта

Объекты Exception поддерживают только файлы программ (PRG) (листинг 14.4).

Листинг 14.4. Демонстрационный пример вложенной конструкции TRY... CATCH... ENDTRY и THROW

```
try
  try
    wait window xxx
  catch to loException when loException.ErrorNo = 1
    wait window 'Error #1'
  catch to loException when loException.ErrorNo = 2
    wait window 'Error #2'
  catch to loException
```

```
    throw loException
  endtry
catch to loException
  messagebox('Error #' + transform(loException.ErrorNo) + chr(13) + ;
    'Message: ' + loException.Message, 0, 'Thrown Exception')
  messagebox('Error #' + transform(loException.UserValue.ErrorNo) + chr(13) + ;
    'Message: ' + loException.UserValue.Message, 0, 'Original Exception')
endtry
```

Выход из *TRY...CATCH...FINALLY*

Для немедленного прекращения блока команд в декларациях TRY или CATCH можно использовать команду EXIT. В этом случае программа перейдет к исполнению блока кода декларации FINALLY, а если его нет, то строки кода, следующей за ENDTRY.

Использование команд в структуре *TRY...CATCH...FINALLY*

В блоки команд деклараций TRY и CATCH можно включать различные команды.

Существуют команды, использование которых в этих блоках не разрешается. В случае использования неразрешенных команд Visual FoxPro генерирует ошибку в режиме run time. В табл. 14.4 приведены команды и указана возможность их применения в блоках TRY, CATCH и FINALLY.

Таблица 14.4. *Применение команд в блоках TRY, CATCH и FINALLY*

Команда	Исполнение в TRY	Исполнение в CATCH	Исполнение в FINALLY
CANCEL	Да	Да	Да
CLEAR ALL	Да	Нет	Нет
CLOSE ALL	Да	Да	Да
DOEVENTS	Да	Да	Да
ERROR	Да	Да	Да
EXIT	Да	Да	Да
LOOP	Нет	Нет	Нет
QUIT	Да	Да	Да
RELEASE	Да	Да	Да
RESUME	Да	Да	Да
RETRY	Нет	Нет	Нет
RETURN	Нет	Нет	Нет
RETURN TO MASTER	Нет	Нет	Нет
RETURN TO	Нет	Нет	Нет
SET STEP ON	Да	Да	Да

SUSPEND	Да	Да	Да
THROW	Да	Да	Да

Такие команды, как LOOP, можно включать в блок только тогда, когда для них имеются внешние циклы FOR EACH...ENDFOR или DO WHILE... ENDDO. Команда ERROR, включенная в любой из блоков, обрабатывается так же, как и любое другое исключение или использование команды THROW.

Исполнение завершающих команд

Если блок FINALLY определен, то выполняется содержащийся в нем код и очищаются все ресурсы, выделенные блоку TRY. Блок FINALLY исполняется всегда, когда происходит исключение — необработанное или обработанное другим обработчиком. Если необходимо обойти выполнение блока FINALLY, нужно написать код в начале блока FINALLY, определяющий какое-либо условие, при выполнении которого программа покидает блок FINALLY.

Обработчик события *Error*

Один из возможных способов обработки ошибки — это обработка с помощью функции-обработчика Error. Рассмотрим такой пример (листинг 14.5).

```
lobj=NewObject("lcfClass")
lobj.NewMethod
DEFINE CLASS lcfClass as Custom
Procedure NewMethod
hd=FOPEN("c:\sys1400.sys")
IF hd<=-1
   DO error
endif
endproc
Procedure Error(nError, cMethod, nLine)
? "Error: ",nError, cMethod, nLine
endproc
ENDDEFINE
```

Если обработчик Error отсутствует, то ошибка будет передана стандартному обработчику VFP либо обработчику, указанному в команде ON ERROR. При наличии нескольких обработчиков и возникновении ошибки в методе или событии объекта предпочтение будет оказано обработчику событий объекта. Если ошибка произойдет вне метода или события, она будет обработана ON ERROR. Если метод Error перегружен, то

он имеет приоритет перед всеми остальными обработчиками прерываний (ON ERROR, TRY...ENDTRY).

Трассировка. Использование точек останова отладчика

Для обнаружения сложных ошибок, в том числе ошибок в логике программы, можно использовать отладчик Visual FoxPro. Отладчик — это мощное средство тестирования приложений. С его помощью вы сможете:

- просматривать ход выполнения программы в пошаговом режиме;
- устанавливать точки останова;
- просматривать и изменять значения переменных.

Точки останова наиболее часто используются при отладке сложных конструкций. Количество их в программе не регламентируется, поэтому программист может задать несколько точек, в которых выполнение программы будет приостановлено и управление передано отладчику. Останов происходит на строке, предшествующей точке останова, и слева рядом со строкой появляется крупная яркая точка. Если в качестве точки останова выбрано событие, то останов произойдет после совершения данного события. Методика установки точки останова проста: на нужной строке кода необходимо дважды нажать левую кнопку мыши. Второй способ состоит в том, чтобы установить курсор на нужной строке кода и нажать кнопку **Toggle breakpoint** на панели инструментов.

Рассмотрим работу отладчика на примере (рис. 14.1).

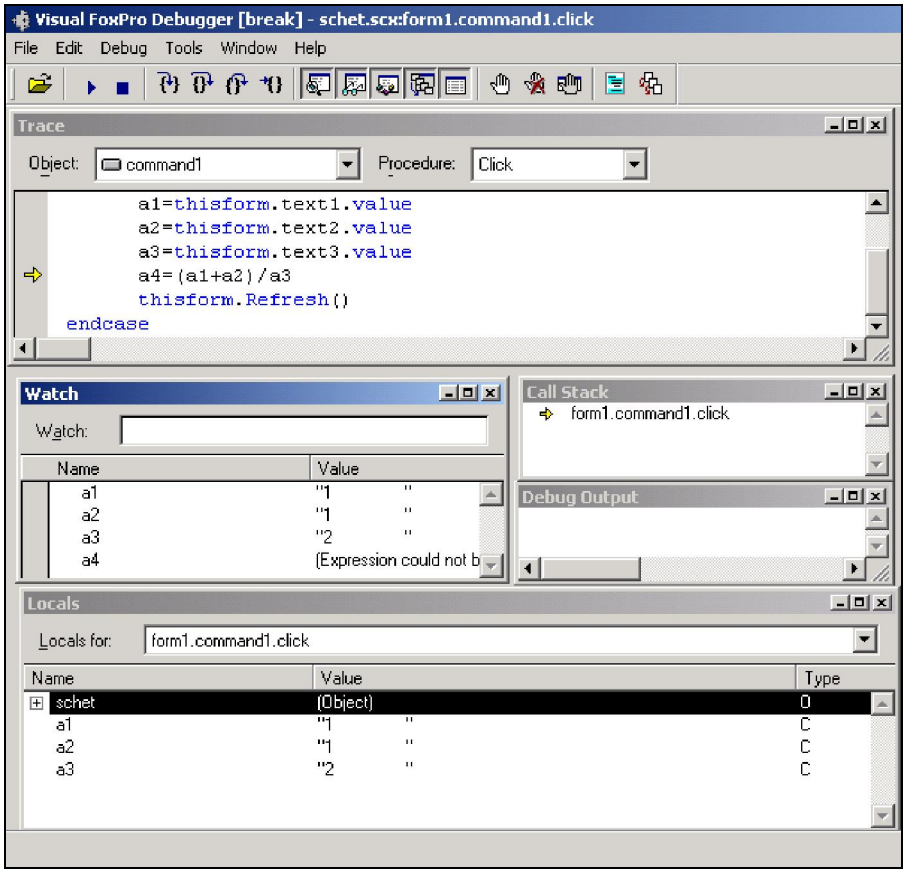

Рис. 14.1. Пример работы отладчика

Наша программа производит некоторые вычисления, данные для которых вводит оператор. В момент вычисления происходит ошибка. Чтобы определить причину ошибки, используем отладчик. В окне **Trace** мы видим фрагмент программного кода, при этом указатель находится на строке с ошибкой. После остановки программы мы можем:

- просмотреть код выше или ниже указателя, перемещаясь с помощью горизонтальной полосы прокрутки;
- выбрать другой объект для просмотра, воспользовавшись списком объектов в окнах **Trace** или **Locals**;
- открыть окно **Data session** и посмотреть открытые таблицы, их индексы и даже записи;
- посмотреть и изменить при необходимости значения переменных памяти или элементов массивов;

- запустить программу на дальнейшее выполнение, нажав кнопку **Resume** ▸;
- прекратить выполнение, нажав кнопку **Cancel** ■.

В окне **Locals** в момент приостановки программы появляются все определенные на данный момент в текущей подпрограмме или функции переменные, а также их значения и типы. Кроме того, тут же представлены переменные — ссылки на объект. Нажав на значок "+" перед именем переменной-ссылки, можно развернуть объект и посмотреть значения его свойств. В нашем случае мы видим, что переменные `a1`, `a2`, `a3` имеют символьный тип данных, а мы пытаемся произвести с ними арифметическую операцию (деление), которая запрещена для символьных данных.

Те же самые переменные показаны и в окне **Watch**. В этом окне можно посмотреть не только переменные, но и значение выражения, в которое входят эти переменные. Для этого в поле **Watch** нужно ввести анализируемое выражение. Если оно не может быть вычислено, то в колонке **Value** появится сообщение "**Expression could not be evaluated**" ("Значение выражения не может быть вычислено"). Точка останова может быть установлена у любой переменной или выражения в окне **Watch**, в этом случае останов произойдет при любом изменении значения отмеченной переменной или выражения.

Точно так же, как в окне **Trace** указатель помечает выполняемую в данный момент строку, в окне **Call Stack** стрелка указывает на текущую выполняемую подпрограмму.

Это окно бывает крайне необходимо, когда нужно отследить последовательность подпрограмм, которая привела к ошибочной ситуации.

В окне **Debug Output** выводится строка символов, определенная командой `DEBUGOUT <выражение>`. Включив в программу эту команду, вы можете прослеживать сообщения, облегчающие понимание того, на какой стадии обработки находится программа.

Coverage Profile

Это отдельное приложение, которое позволяет анализировать ход выполнения программы. Для анализа нужно выполнить следующую последовательность действий:

- создать файл с расширением log, в который будет записываться информация о выполнении программы;
- запустить выполняемую программу или приложение;
- проанализировать файл с расширением log с помощью программы **Coverage Profiler**.

1. Задаем log-файл. Это можно сделать двумя способами.
   - Использовать следующую команду:

     ```
     SET COVERAGE TO имя_файла_протокола.log [ADDITIVE]
     ```

При использовании предложения `ADDITIVE` новая информация добавляется к существующему файлу *`cCoverage`*`.log`. Если фраза `ADDITIVE` отсутствует, информация в созданном файле обновляется. Выполнение команды `SET COVERAGE TO` без аргумента прекращает передачу информации в файл протокола.

- В меню **Tools** отладчика (Debugger) выбрать пункт **Coverage Logging**.

В появившемся диалоговом окне нужно ввести имя log-файла:

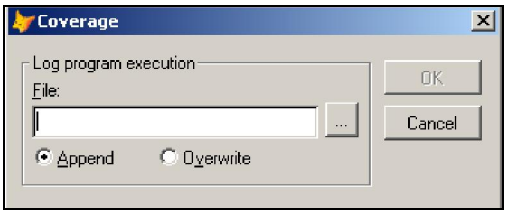

Рис. 14.2. Диалоговое окно **Coverage**

2. Запуск приложения.

   - Первый способ: запустить команду

     ```
     DO (_COVERAGE) [WITH <имя файла протокола.log>]
     ```

   Если имя файла протокола не указано, то Visual FoxPro запрашивает имя файла в режиме диалога.

   - Второй способ: использовать команду меню **Tools | Coverage Profiler**.

   После запуска окно разделится на три панели (рис. 14.3).

   В левой верхней панели отображаются объекты. В правой верхней панели определен путь к классам соответствующих объектов. В нижней панели отображаются методы объекта, к которым происходило обращение в процессе выполнения. Вертикальные пометки слева от некоторых строк кода указывают на то, что этот код еще ни разу не выполнялся. Можно настроить пометки наоборот, т. е. таким образом, что они будут указывать на выполненный код, а не пропущенный. Настройка происходит в окне **Options Coverage Profiler** (рис. 14.4).

3. Анализ работы программы.

   Файл протокола (LOG) можно использовать для просмотра статистической информации. Для того чтобы открыть диалоговое окно **Coverage Profiler Statistics**, используйте путь **Tools | Coverage Profiler | Statistics** (рис. 14.5).

Для просмотра статистики нажмите кнопку **Statistics by Project**.

Для просмотра файла протокола нажмите кнопку **Source Text Log**.

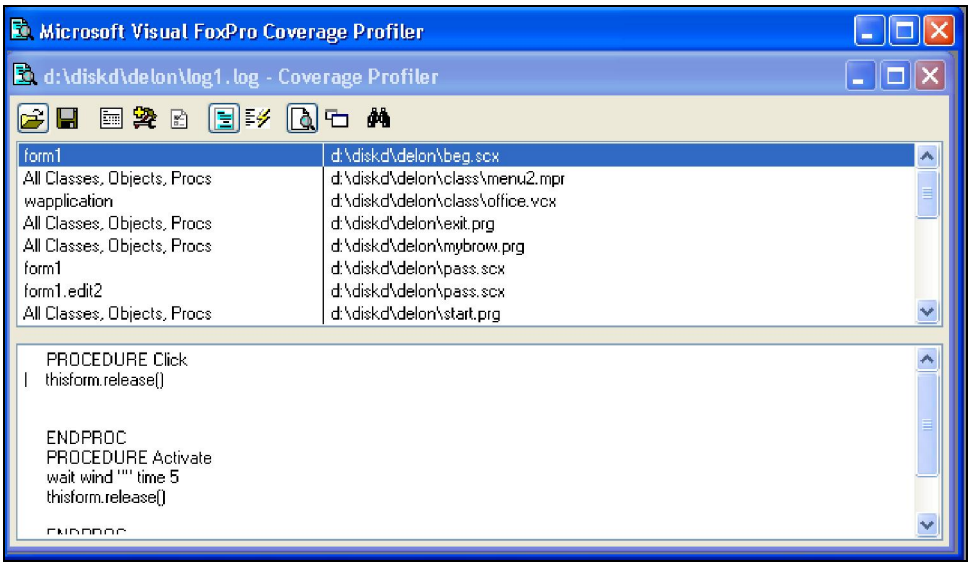

Рис. 14.3. Окно **Coverage Profiler** во время запуска программы

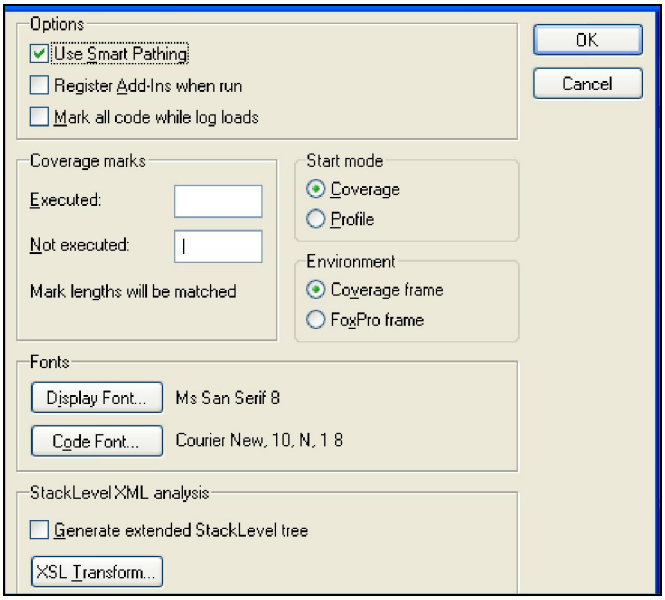

Рис. 14.4. Окно настройки **Options Coverage Profiler**

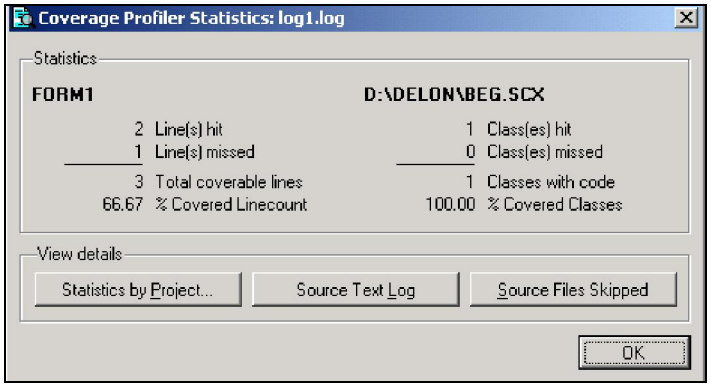

Рис. 14.5. Окно диалога **Coverage Profiler Statistics**

Чтобы увидеть список файлов, не включенных в файл протокола, нажмите кнопку **Source Files Skipped**.

Файл протокола состоит из записей, разделенных запятыми, и имеет следующий вид (рис. 14.6).

В табл. 14.5 приведена структура записей log-файла.

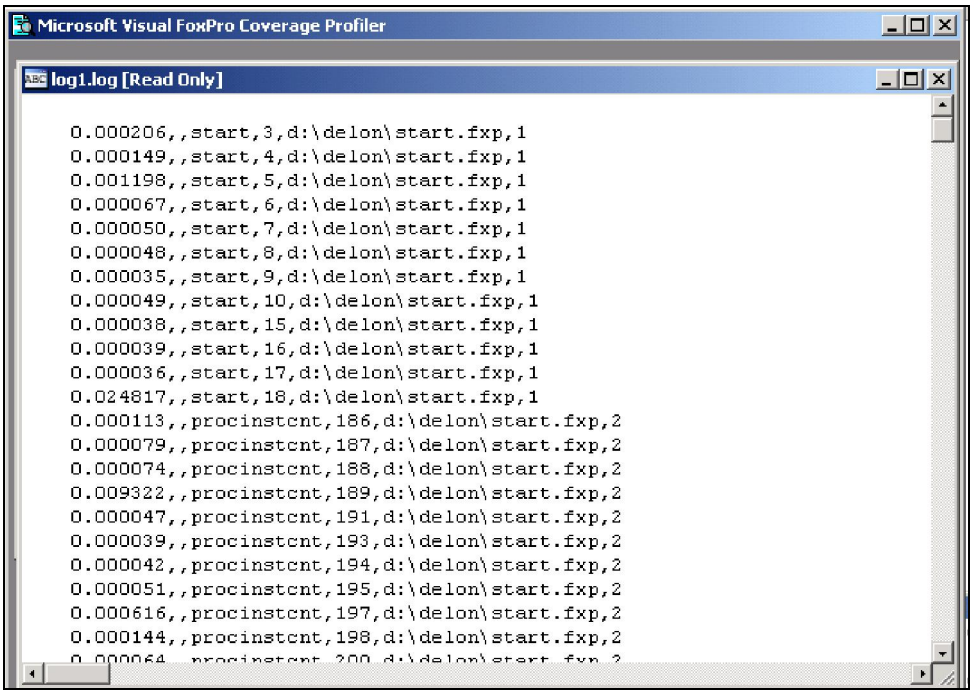

Рис. 14.6. Просмотр файла протокола (LOG)

Таблица 14.5. *Структура записей log-файла*

Номер	Описание
1	Время выполнения кода строки
2	Имя класса, содержащего выполняемый код программы
3	Метод, объект или программа, в которых расположен выполняемый код
4	Номер строки в методе или программе
5	Полное наименование файла
6	Уровень вложенности вызываемой программы