О.В. Бартеньев

Современный Фортран

Издание третье, дополненное и переработанное

МОСКВА «ДИАЛОГ-МИФИ» 2000

УДК 519.682
ББК 32.97
Б24

Бартеньев О. В.
Б24 Современный Фортран. - 3-е изд., доп. и перераб. - М.: ДИАЛОГ-МИФИ, 2000. - 449 с.

ISBN 5-86404-113-0

Излагаются базовые свойства Фортрана фирм Compaq и Microsoft, основанные на стандартах Фортран 90 и 95. По существу, пособие является новой, усовершенствованной версией одноименного издания 1998 г. Дополнительно в книгу включены нововведения стандарта Фортран 95, рассмотрены отличительные особенности Фортрана Compaq и описаны методы использования объектов ActiveX.

Как и ранее, пособие ориентировано как на пользователей со стажем, так и на начинающих программистов, для которых разбираются методы программирования и особенности их реализации на Фортране.

Предназначено для студентов, преподавателей, аспирантов, инженеров и научных работников.

Учебно-справочное издание
Бартеньев Олег Васильевич
Современный Фортран.

Предисловие

Новое издание пособия обусловлено прежде всего расширяющимся интересом российских специалистов к современному Фортрану, и в частности к его реализациям фирмами Compaq и Microsoft, известными как Compaq Visual Fortran 6.1 (далее - CVF) и Microsoft Fortran PowerStation 4.0 (далее - FPS).

Более продвинутой разработкой является CVF. Это неудивительно, поскольку Microsoft вот уже несколько лет не поддерживает FPS и он живет своей одинокой жизнью. В то же время CVF, включая все возможности FPS, постоянно совершенствуется и развивается. В дополнение к FPS, в Фортране Compaq реализованы стандарт 1995 г. и большое число дополнительных возможностей, в том числе новая версия математической библиотеки IMSL (IMSL Fortran 90 MP), конструктор модулей для объектов ActiveX, визуализатор массивов и др.

Тем не менее, учитывая широкое распространение FPS среди российских пользователей, в книге рассматриваются и CVF и FPS. Впрочем, такое объединение не потребовало чрезмерных усилий, так как, во-первых, CVF наследует все черты FPS: практически все, что работает в FPS, будет работать и в CVF (определенные различия имеются в организации передачи данных), и, во-вторых, обе разработки эксплуатируют одну и ту же среду - Microsoft Developer Studio (далее - DS).

Пособие содержит описание базовых, основанных на стандартах Фортран 90 и 95 свойств языка и имеющихся расширений, таких, как целочисленные указатели, объединения, целочисленные выражения в спецификаторе формата и др. Из дополнительных возможностей приводятся процедуры библиотеки DFLIB (MSFLIB), позволяющие, например, управлять файлами или операциями с плавающей точкой, а также рассматривается конструктор модулей - новое средство CVF, осуществляющее генерацию модулей, облегчающих использование в приложениях Фортрана объектов ActiveX, т. е. объектов, предоставляемых другими приложениями и поддерживающих модель многокомпонентных объектов COM. Например, Excel предоставляет множество иерархически связанных объектов, таких, как "Рабочая книга", "Рабочий лист" или "Диаграмма".

Большое число иных, не предусмотренных стандартом специальных средств рассматривается в книгах [1-3], изучая которые пользователь получает возможность создавать диалоги, меню, обрабатывать события, выполнять многооконный, средствами QuickWin или OpenGL, графический вывод, писать разноязычные, например на Фортране и СИ, приложения, вызывать математические процедуры библиотеки IMSL и т. д.

Однако главное назначение Фортрана - это быстрый счет в различных научно-технических приложениях. (Это та область, в которой у Фортрана нет конкурентов.) Поэтому основной задачей пользователя Фортрана является освоение техники высокопроизводительных вычислений, и поэтому основная цель разработчиков Фортрана - поиск и включение в язык соответствующих высокоскоростных средств. В этом плане прогресс несомненен, и его основой является отраженная в стандартах Фортран 90 и 95 концепция обработки массивов.

В соответствии с ней:

- массивы, как и обычные скалярные переменные, могут быть использованы в выражениях, результатами которых также являются массивы;
- в конструкциях WHERE и FORALL, заменяющих громоздкие циклы, можно под управлением маски выполнять сложные присваивания массивов. Причем для употребления в FORALL введен класс так называемых чистых, создаваемых пользователем процедур;
- введено большое число функций для массивов, реализующих в том числе базовые операции линейной алгебры, например вычисляющих произведение матриц, скалярное произведение векторов или возвращающих транспонированную матрицу;
- большинство встроенных процедур являются элементными, т. е. способными принимать массивы, выполнять поэлементно необходимые вычисления и возвращать массивы. Например, если *x* и *y* - это векторы одного размера, то вызов *y* = SIN(*x*) позволит сформировать вектор *y*, каждый элемент которого равен синусу соответствующего элемента вектора *x*;
- свои собственные элементные процедуры, обладающие такими же, как и встроенные элементные процедуры, свойствами, может создать и программист;
- введены подобъекты массивов, называемые сечениями, которые можно использовать так же, как и массивы (т. е. в выражениях, в качестве параметров элементных функций), и применяя которые можно заменить многие циклы, например неявные циклы операторов ввода/вывода;
- при необходимости можно объявлять динамические (размещаемые, ссылочные, а в процедурах - автоматические) массивы;
- можно реализовать массивоподобные функции, возвращающие массивы или ссылочные динамические массивы.

Эти, далеко не в полной мере перечисленные, относящиеся к массивам нововведения в сочетании с другими усовершенствованиями позволяют создавать быстро работающие, компактные, хорошо читаемые и удобные для употребления программы. Больше того, при наличии многопроцессорной вычислительной системы и соответствующего компилятора языка Фортран применение вышеназванных новых свойств обеспечивает естественное распараллеливание вычислений и, как следствие, существенный рост производительности.

1. Элементы языка

1.1. Свободная форма записи программы

Рассмотрим программу, в которой задаются два действительных числа, вычисляется их сумма и выводится результат:

```
program p1                ! p1 - имя программы
 real x, y, z             ! Объявляем 3 переменные вещественного типа
 x = 1.1                  ! Присваиваем переменным x и y значения
 y = 2.2
 z = x + y                ! Присваиваем z результат сложения x и y
 print *, 'z = ', z       ! Вывод результата на экран
                          ! Результат вывода: z =       3.300000
end program p1            ! END - обязательный оператор завершения программы
```

Приведенная программа называется *главной*. Она построена по схеме: сначала следует объявление типов используемых переменных, затем - операторы, выполняющие над объявленными переменными некоторые действия. Эта схема является типовой и неоднократно воспроизводится в пособии.

Программа завершается оператором END, в котором имя программы *p*1 и одновременно PROGRAM *p*1 могут быть опущены. Иными словами, программу можно завершить так: END PROGRAM или END. Программа имеет заголовок: PROGRAM *имя-программы*. Однако такой заголовок может быть опущен. В этом случае *имя-программы* не может присутствовать в операторе END. *Имя-программы* должно отличаться от используемых внутри главной программы имен.

Программа записана в *свободной форме*. По умолчанию файл с текстом написанной в свободной форме программы имеет расширение F90. В нем не должно быть директивы $NOFREEFORM, а при компиляции нельзя задавать опцию компилятора /4Nf.

Замечание. Здесь и далее указываются присущие FPS опции компилятора. Описание опций компилятора CVF и соответствие между опциями компиляторов FPS и CVF изложены в [1].

В свободной форме текст программы записывается по правилам:

- длина строки текста равна 132 символам;
- запись оператора может начинаться с любой позиции строки;
- на одной строке могут размещаться несколько разделенных точкой с запятой (;) операторов;
- если строка текста завершается символом &, то последующая строка рассматривается как строка продолжения;

- в операторе может быть до 7200 символов. Число строк продолжения в свободной форме не может быть более 54;
- любые расположенные между восклицательным знаком и концом строки символы рассматриваются как комментарий, например:

```
real x, y,                &     ! Комментарий в начальной строке
  z, a(5),                &     ! Строка продолжения
  r, b(10)                      ! Еще одна строка продолжения
x = 1.1; y = 2.2; a = -5.5      ! Операторы присваивания
```

Замечание. Помимо свободной программу можно записать и в *фиксированной*, унаследованной от Фортрана 77 форме (прил. 2). Файлы, содержащие текст в фиксированной форме, по умолчанию имеют расширения F или FOR. В файлах с такими расширениями можно перейти и к свободной форме, задав директиву $FREEFORM или опцию компилятора /4Yf [1].

Запустим теперь программу *p*1, используя приведенные в разд. 1.2 сведения.

1.2. Консоль-проект

Программа рассматривается в FPS и CVF как проект. Для запуска новой программы необходимо прежде всего его создать. Существует несколько типов проектов, однако на первых порах мы будем работать с консоль-проектом - однооконным проектом без графики.

1.2.1. Создание проекта в CVF

Начнем создание проекта с запуска DS. Для этого после запуска Windows 95 или Windows NT можно выполнить цепочку действий: пуск - программы - Compaq Visual Fortran - Developer Studio. Перейдем к созданию нового консоль-проекта CVF. Для этого выполним цепочку File - New - выбрать закладку Projects - Fortarn Console Application - задать имя проекта, например proj1, - задать папку размещения проекта, например D:\FORTRAN\proj1, - OK. В появившемся затем окне выбрать кнопку Anempty project и нажать Finish. Тогда будет создана директория (папка), имя которой совпадает с именем проекта. В этой папке будут размещены файлы проекта с расширениями DSP, DSW и NCB. Также будет создана папка Debug. Сам же проект отобразится на закладке FileView (рис. 1.1).

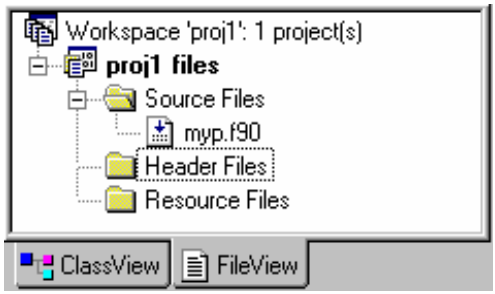


Рис. 1.1. Проект proj1

Создадим теперь файл, в который будет занесен текст программы, выполнив: File - New - выбрать закладку Files - выбрать Fortran Free Format Source File - активизировать опцию Add to project - задать имя файла, например myp (расширение опустить), - OK. Созданный файл получит расширение F90 и разместится в D:\FORTRAN\proj1.

Если же файл уже существует, то для его добавления в проект в окне FileView выберите папку Source Files и выполните: нажать на правую кнопку мыши - Add Files to Folder - выбрать тип файлов и необходимые файлы (рис. 1.2) - OK.

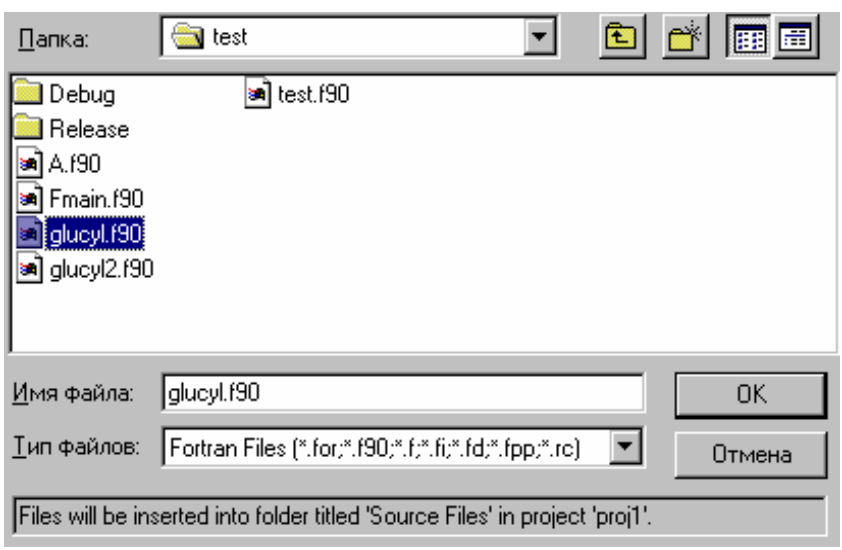


Рис. 1.2. Добавление файлов в проект

Чтобы отобразить на экране содержимое файла, достаточно ударить по нему дважды мышью.

1.2.2. Создание проекта в FPS

FPS может быть оснащен более ранней версией DS, в которой схема создания проекта и добавления в него файла несколько иная.

После запуска DS выполним цепочку File - New - Project Workspace – OK - Console Application - ввести имя проекта - задать расположение проекта на диске - Create. После нажатия кнопки Create будет создана директория (папка), имя которой совпадает с именем проекта. В этой папке будут размещены файлы проекта с расширениями MAK и MDP.

Создадим теперь новый файл, выполнив File - New - Text File - OK. Наберем далее в правом окне текст программы и запишем его на диск: File - Save - выбрать на диске директорию для записи файла - задать имя файла с расширением, например myp.f90, - сохранить.

Добавим созданный файл в проект: Insert - File Into Project - выбрать файл myp.f90 - Add.

1.2.3. Операции с проектом

Чтобы закрыть проект, следует выполнить: File - Close Workspace. Существующий проект открывается в результате выполнения цепочки File - Open Workspace - выбрать файл проекта - Open. Для удаления файла из открытого проекта достаточно выбрать этот файл в окне FileView и нажать Del.

Выполним теперь компиляцию проекта: Build - Compile - и исправим обнаруженные ошибки, сообщения о которых вы найдете в нижнем окне на закладке Build.

Создадим выполняемый EXE-файл: Build - Build. Запустим его для исполнения: Build - Execute - и получим результат. Для выхода из рабочего DOS-окна, в котором отобразились результаты, нажмем любую клавишу, например Esc или Enter.

Компиляцию, сборку и запуск программы можно также выполнить, используя имеющиеся в среде кнопки (Compile, Build, GO) или выбирая на клавиатуре соответствующие сочетания клавиш, информация о которых размещается в пунктах меню DS. Здесь же отметим, что все три упомянутых выше этапа (компиляция, сборка и запуск) будут выполнены после нажатия Ctrl+F5. Впрочем, если в проекте несколько файлов, удобнее (с позиции выявления ошибок) компилировать файлы по отдельности, переходя к обработке следующего файла после устранения всех синтаксических ошибок в текущем.

1.2.4. Файлы с исходным текстом

В общем случае файлы с исходным текстом программы могут иметь расширения F90, F и FOR. Например, myp.f90, myp.f, myp.for. В первом случае компилятор считает, что файл написан в свободной форме. В двух последних по умолчанию предполагается, что исходный текст записан в фиксированной форме (прил. 2). Мы же будем использовать для файлов расширение F90 и свободную форму записи исходного текста.

1.3. Операторы

Написанная на Фортране программа - это последовательность операторов языка программирования. Операторы разделяются на *выполняемые* и операторы, которые не участвуют в вычислениях и называются *невыполняемыми*.

Выполняемый оператор описывает действия, которые должны быть произведены программой.

Невыполняемые операторы описывают элементы программы, например данные или программные компоненты.

Наиболее часто используется выполняемый оператор присваивания, имеющий вид:

имя переменной = выражение

В результате его выполнения *переменной* присваивается результат некоторого *выражения*. Например:

```
real :: d, a = 1.2          ! Невыполняемый оператор объявления
                            ! типа данных, в котором переменная a
                            ! получила начальное значение 1.2
d = 2.3                     ! Читается: d присвоить 2.3
a = a + 4.0 * sin(d)        ! Значение a изменится с 1.2 на 4.182821
print *, a                  ! Вывод значения переменной a
end                         ! Оператором END завершаем программу
```

Замечание. Оператор присваивания будет лучше читаться, если перед и после знака оператора = поставить по одному пробелу.

Все используемые в программе объекты данных, например переменные, следует предварительно объявить, т. е. явно указать их тип и при необходимости другие свойства. Для этих целей существуют невыполняемые операторы объявления типа, например:

```
real x, y       ! Невыполняемый оператор REAL объявляет две переменные
                ! x и y вещественного типа
integer k       ! Невыполняемый оператор INTEGER объявляет переменную
                ! k целого типа, принимающую целые положительные
```

```
                    ! и отрицательные значение и нуль, например:
k = -55
```

Невыполняемые операторы объявления типа должны располагаться в программе ранее любого исполняемого оператора.

1.4. Объекты данных

Программа выполняет обработку данных. Данные представлены в программе в виде *переменных* и *констант*. *Объекты данных* (переменные и константы) различаются именами, типами и другими свойствами. Переменная, имя которой присутствует в программе, считается *существующей*. Существующая переменная может быть *определенной* и *неопределенной*. Переменная становится определенной после того, как она получит значение, например в результате присваивания или выполнения ввода. Константы бывают *именованными* и *буквальными* (*неименованными*). Именованная константа объявляется с атрибутом PARAMETER. Значение именованной константы не может быть изменено в результате вычислений. Поэтому ее имя не может находиться в левой части оператора присваивания или быть элементом списка ввода.

```
real a, b           ! Объявляем вещественные переменные с именами a и b
                    ! Задание именованной константы n
integer, parameter :: n = 5
                    ! Все именованные константы имеют атрибут PARAMETER
a = 4.5             ! Теперь переменная a определена, ей присвоено
                    ! значение буквальной константы 4.5
read *, b           ! После ввода будет определена переменная b
```

Замечание. При записи не имеющей десятичных знаков вещественной константы следует использовать десятичную точку, например:

```
real a
a = 4           ! Так записывать не следует
a = 4.0         ! Такая запись подчеркивает тип используемых данных и
                ! не требует дополнительных преобразований типов данных
```

Начальное значение переменной может быть установлено оператором объявления типа или оператором DATA. В случае задания начальных значений (посредством присваивания) или атрибутов оператор объявления типа должен содержать разделитель :: .

```
real :: a = 1.2, b, c          ! Разделитель :: необходим
real d / 4.5 /                 ! Разделитель :: может быть опущен
data b, c / 1.5, 4.8 /         ! или: data b / 1.5 /, c / 4.8 /
```

В приводимых выше примерах переменные содержат одно значение. Такие переменные называются *простыми*. Однако можно задать *составные*

переменные, содержащие более одного значения. Примером такой переменной является *массив*. Используя имя составной переменной, можно обеспечить доступ сразу к нескольким значениям. Например:

```
real a(5)                ! Объявляем вещественный массив a из пяти элементов
a(1) = 1.2               ! a(1) - имя первого элемента массива a
a(2) = 1.3               ! Присвоим значение 1.3 второму элементу массива a
a(3) = 1.4; a(4) = -4.2; a(5) = 0.0
print *, a               ! Вывод всех элементов массива a
                         ! Следующий вывод эквивалентен PRINT *, a
print *, a(1), a(2), a(3), a(4), a(5)
end
```

Массив не может иметь в качестве элементов другие массивы.

Рассмотренный в примере массив является одномерным. Могут быть заданы и многомерные (с числом измерений не более семи) массивы. Протяженность каждого измерения массива задается нижней и верхней границами, которые разделяются двоеточием. Если нижняя граница равна единице, она может быть опущена. В этом случае опускается и разделяющее двоеточие. Например, каждое из следующих объявлений задает массив из 10 элементов:

```
real a(-4:5), b(0:9), c(1:10), d(10)
```

Число измерений массива называется его *рангом*. Объект данных, ранг которого равен нулю, называется *скаляром*.

В процессе вычислений *значение переменной* может быть определено или изменено, например, операторами ввода или присваивания. Есть ситуации, в которых значение переменной может стать неопределенным. Такой объект данных, как массив, считается неопределенным, если не определен хотя бы один из его элементов.

Для определения массива или его изменения можно использовать *конструктор массива*. Он может быть применен и в операторах объявления типа, и среди исполняемых операторов, например:

```
real :: a(5) = (/ 1.1, -2.1, 3.1, -4.5, 5.0 /)
real b(5)
b = (/ 1.1, -2.01, 3.1, 4.05, 50.0 /)
```

Объекты данных различаются типом. Возможные *типы данных*: целый, вещественный, комплексный, логический, символьный и производный тип - структуры. Элементами массива могут быть объекты одного типа.

Примеры объявления объектов данных разных типов:

```
real :: c = 4.56, b(20)          ! c и b  - вещественные переменные
complex :: z = (1.4142, 1.4142)  ! z - переменная комплексного типа
```

```
character(30) :: fn = 'c:\dig.bin'        ! fn - символьная переменная
real, parameter :: pi = 3.141593          ! pi - вещественная константа
```

В программе составной объект может быть использован целиком, можно также использовать и часть составного объекта, которая называется *подобъектом*. Так, в случае массива его подобъектом является отдельный элемент массива, а также и любая часть массива, которая называется *сечением массива*. Например:

```
real a(5)                 ! Объявляем вещественный массив a из пяти элементов
a = 1.2                   ! Присвоим значение 1.2 всем элементам массива a
a(2) = -4.0               ! Присвоим значение -4.0 второму элементу массива
print *, a(1:3)           ! Вывод сечения массива a - первых трех его элементов
print *, a(1), a(2), a(3) ! Этот вывод эквивалентен предыдущему
```

1.5. Имена

Переменные, константы, программные компоненты имеют *имена*. Имя - это последовательность латинских букв, цифр, символа $ или подчеркивания, начинающаяся с буквы или символа $. Имя не должно содержать более 31 символа. Регистр букв не является значащим. Так, имена st, St, sT, ST есть одно и то же. Следует придумывать имена, отображающие смысл применяемых переменных, констант и других объектов программы.

Примеры имен: Cat1 F_Name $var stlen

Имена разделяются на *глобальные*, например имя главной программы или встроенной процедуры, и *локальные*, например имя переменной или константы.

Разрешается создавать локальные имена, совпадающие с глобальными именами встроенных процедур. Но если в программной единице имя, например *sum*, использовано для имени локальной переменной, встроенная функция SUM в этой программной единице будет недоступна. Поэтому для создаваемых объектов следует придумывать имена, которые отличаются от имен встроенных процедур. Не стоит также давать создаваемым объектам имена, совпадающие с именами операторов и других объектов Фортрана.

Замечание. Широко используется при формировании имен так называемая *Венгерская нотация*. В соответствии с ней имя объекта снабжается префиксом из строчных букв, указывающих его тип (если объект обладает типом). Последующая часть имени раскрывает его смысл. Причем каждая часть имени, отражающая отдельный смысловой компонент, начинается с прописной буквы. Например, имя iVectorSize может быть дано именованной константе, хранящей размер вектора, а имя SetInitValues может иметь подпрограмма, выполняющая инициализацию переменных. В

примерах пособия Венгерская нотация (за редкими исключениями) не употребляется.

1.6. Выражения и операции

Выражение - это формула, по которой вычисляется значение, например 2.0**cos*(*x*/4.5). Выражение состоит из операндов и нуля или более операций. Используемые в выражениях *операции* разделяются на *двуместные* и *одноместные* (*унарные* + и -). В двуместной операции участвуют два операнда, в одноместной - один. Например:

```
b + c                    ! Простое выражение с двуместной операцией
-b                       ! Простое выражение с одноместной операцией
c                        ! Выражение без операций
```

Операндами выражения могут быть константы, переменные и вызовы функций. Выражение может присутствовать в правой части оператора присваивания, в операторах вывода, в вызовах процедур и других операторах языка. Общий вид выражения, в котором присутствуют двуместные операции:

операнд *операция* операнд *операция* операнд ...

Значение каждого операнда выражения должно быть определено, а результат должен иметь математический смысл. Например, не должно быть деления на нуль.

Замечание. Подобъект составного объекта также является переменной и, следовательно, может быть операндом выражения. Например:

```
real a(10) = 3.0, b(7)          ! Массив является составной переменной
a = 2.0 * a                     ! Массив как элемент выражения
! Элементом выражения является сечение подобъект массива - его сечение a(2:8)
b = a(2:8) / 2.5
```

В зависимости от типа возвращаемого результата выражения подразделяются на *арифметические*, *логические*, *символьные* и *производного типа*. Для выражений первых трех типов в Фортране определены встроенные операции. В выражениях производного типа операции должны быть заданы программистом. Встроенные арифметические операции приведены в табл. 1.1.

Таблица 1.1. Встроенные арифметические операции

Действия	*Обозначения*	*Примеры*	*Запись на Фортране*
Возведение в степень	**	$\sqrt[3]{2}$	2**(1.0 / 3.0)
Умножение, деление	*, /	a × b; a : b	a * b; a / b

Сложение, вычитание	+, -	a + b; a - b	a + b; a - b
Унарные	+ и -	+2; -5.5	+2; -5.5

Пример арифметического выражения:

```
real :: a = -1.2
a = a * a + 2.2**2                ! Возвращает 6.28
```

Возведение в степень имеет ограниченную область действия. Так, запрещается возводить отрицательное число в нецелочисленную степень, например ошибочно выражение (-2)**3.1. Также нельзя возводить нуль в отрицательную или нулевую степень.

Операции различаются *старшинством*, или *приоритетом*. Среди арифметических операций наибольшим приоритетом обладает операция возведения в степень, далее с одинаковым приоритетом следуют умножение и деление, одинаковый и наименьший приоритет имеют сложение, вычитание и унарные + и -. Например, -3**2 возвращает -9, а не 9.

Выражения без скобок вычисляются слева направо последовательно для операций с одинаковым приоритетом, за исключением операций возведения в степень, которые выполняются справа налево. Если требуется изменить эту последовательность, то часть выражения, которую нужно вычислить в первую очередь, выделяется скобками. Иногда скобки используются и для улучшения читаемости выражения. Между элементом выражения и знаком операции для улучшения читаемости выражения можно проставить один пробел.

Пример:

```
real :: a, c, d = 1.1
real :: s1 = -1.0, s2 = -2.2, s3 = 3.3
d = (d + 5.17) / 46.2             ! Прежде вычисляется выражение в скобках
a = d - (s1 + s2 + s3)            ! или a = d - s1 - s2 - s3
c = 2.0**2.0**(1.0/3.0)           ! Отобразим порядок вычислений,
c = 2.0**( 2.0**(1.0/3.0) )       ! расставив скобки
```

Знак одноместной операции не должен следовать непосредственно за другим знаком операции. Чтобы этого избежать, подвыражение с одноместной операцией заключается в скобки. Например:

```
a = 4 / -2                        ! Ошибка
a = 4 / (-2)                      ! Правильно
```

Всегда надо учитывать эффект целочисленного деления, при котором отбрасывается получаемая в результате арифметического деления дробная часть, например:

```
-5 / 2                            Возвращает -2
 5 / 2                            Возвращает  2
```

Результатом арифметического выражения может быть целое, вещественное или комплексное число. Результатом логического выражения является либо .TRUE. - *истина*, либо .FALSE. - *ложь*. Результатом символьного выражения является последовательность символов, которая называется *символьной строкой*.

Примеры логического и символьного выражений:

```
real :: a = 4.3, d = -5.0
logical :: fl = .false.                ! Объявление логической переменной
character(10) :: st, st2*3 = 'C6'      ! Объявление символьных переменных st и st2
fl = .not. fl .and. a > d              ! .TRUE.
st = st2 // '- 97'                     ! C6 - 97
```

Выражение является *константным*, если оно образуется из констант. Такого рода выражения могут быть использованы, например, при объявлении массивов или символьных данных:

```
integer, parameter :: n = 20, m = 3 * n
real a(n), d(2*n), c(m)
character(len = n / 2) st
```

Операндами арифметических, логических и символьных выражений могут быть согласованные массивы или их сечения. Одномерные массивы согласованы, если они имеют одинаковое число элементов. Всегда согласованы массив и скаляр, т. е. объект данных, не являющийся массивом. Например:

```
integer :: a(0:4) = 3, b(-1:3) = 7, d(5)
d = (a + b) / 2                        ! Элементы массива d:  5, 5, 5, 5, 5
```

В приведенном примере поэлементно выполняется сложение соответствующих элементов массивов *a* и *b*, затем скаляр 2 расширяется до одномерного массива из пяти элементов, каждый элемент которого равен двум и на который поэлементно делится полученный ранее массив сумм, т. е. оператор $d = (a + b)/2$ эквивалентен оператору

```
d = (a + b) / (/ 2, 2, 2, 2, 2 /)
```

Фортран позволяет использовать в выражениях не только встроенные, но и задаваемые программистом операции. Такие операции применяются, например, при работе с производными типами данных, для которых не определено ни одной встроенной операции. Могут быть заданы как двуместные, так и одноместные операции.

1.7. Присваивание

Оператор присваивания обозначается знаком равенства (=) и записывается в виде

varname = выражение

В результате присваивания переменная *varname* получает новое значение, которое возвращается в результате вычисления *выражения*.

Знак равенства оператора присваивания трактуется иначе, чем знак равенства в математике. Так, в математике запись $k = 2 * k + 1$ означает запись уравнения, решением которого является $k = -1$, а уравнение $k = k + 1$ вообще не имеет решения. В то же время в программе

```
integer :: k = 4
k = k + 1                    ! После присваивания k равно пяти
k = 2 * k + 1                ! После присваивания k равно 11
```

встроенный оператор присваивания определен для числовых, логического и символьного типов данных. Использовать *varname* для переменной производного типа можно, если *выражение* имеет тот же тип, что и *varname*.

Если тип переменной *varname* отличается от типа выражения, то результат выражения преобразовывается в тип *varname*. Поскольку в результате преобразований возможна потеря точности, то необходимо следить, чтобы эта потеря не привела к серьезному искажению результата, например:

```
integer n
real x, y
n = 9.0 / 2          ! После присваивания n равно четырем
x = 9.0 / 2          ! После присваивания x равно 4.5
y = n * 5            ! Возвращает 20 - потеря точности
y = x * 5            ! Возвращает 22.5 - вычисления без потери точности
```

1.8. Простой ввод/вывод

При вводе с клавиатуры данные из текстового представления преобразовываются во внутреннее. При выводе на экран данные из внутреннего представления преобразовываются во внешнее (текстовое). Преобразования ввода/вывода (В/В) можно задать дескрипторами преобразований. Можно также использовать В/В, в котором преобразования выполняются в соответствии с установленными по умолчанию правилами. Такого рода преобразования обеспечиваются *управляемым списком* В/В, который мы и будем преимущественно использовать в поясняющих примерах. Управляемые списком операторы ввода с *клавиатуры* и вывода на *экран* выглядят так:

```
READ(*, *) список ввода          ! Ввод с клавиатуры
READ *, список ввода             ! Ввод с клавиатуры
WRITE(*, *) список вывода        ! Вывод на экран
```

PRINT *, *список вывода* ! Вывод на экран

Список ввода - часть оператора ввода, устанавливающая величины, которые надо ввести.

Список вывода устанавливает величины, которые надо вывести.

Список вывода может содержать произвольные выражения; список ввода - только переменные.

Последняя или единственная звездочка приведенных операторов означает, что В/В управляется списком. В операторах, содержащих две заключенные в скобки и разделенные запятой звездочки, первая - задает устройство В/В (клавиатуру и экран).

Пример:

```
integer n
real a(500)
print *, 'Введите n '          ! На экране появится сообщение: Введите n
read *, n                      ! Используем для В/В циклический список
read *, (a(i), i = 1, n)       ! Потребуется ввести с клавиатуры n значений
print *, (a(i), i = 1, n)      ! Контрольный вывод на экран
```

Выполним ввод:

```
3                              (После ввода значения для n нажимаем Enter)
1 2 3                          (Отдельные значения разделяются пробелом)
```

Результат вывода:

```
1.00000      2.000000      3.000000
```

Замечания:

1. В качестве разделителя задаваемых при вводе значений можно использовать и запятые или запятые вместе с пробелами, например:

```
1, 2, 3
```

2. В вышеприведенном операторе PRINT указан русский текст. Однако если не принять специальных мер, то в DOS-окне, в которое этот текст направляется при работе с консоль-проектами, будут выведены совсем другие символы:

```
-тхфшЄх n
```

Чтобы избежать таких искажений, необходимо в программе, выводящей русский текст в DOS-окно, выполнить ссылку

```
use TextTransfer
```

обеспечить доступ к модулю TextTransfer и выполнить перекодировку текста, применив, имеющуюся в модуле TextTransfer функцию RuDosWin, записав вместо

```
print *, 'Введите n '
```

оператор

```
print *, trim(RuDosWin('Введите n ', .false.))
```

Текст модуля TextTransfer и правила его употребления приведены в прил. 1. В дальнейшем, однако, при использовании русского текста в операторе вывода обращение к RuDosWin явно выполняться не будет, но всегда будет подразумеваться.

1.8.1. Некоторые правила ввода

Для рассмотрения правил ввода введем ряд понятий.

Запись текстового последовательного файла - строка символов, завершаемая символом новой строки.

Поле записи файла - часть записи, содержащая данные, которые могут быть использованы оператором ввода.

Файл состоит из записей и завершается специальной записью *"конец файла"*. При вводе с клавиатуры при необходимости можно проставить запись "конец файла", нажав Ctrl + Z, например:

```
integer ios
real x
do                                  ! Цикл продолжается до нажатия Ctrl + Z
 print '(1x, a, $)', 'Enter x '     ! Вывод подсказки; выполняется без продвижения
 read(*, *, iostat = ios) x         ! После Ctrl + Z будет введена запись "конец файла",
 if(ios == -1) exit                 ! ios примет значение -1 и произойдет выход из цикла
 print *, 'x = ', x
end do
```

Ввод под управлением списка выполняется по правилам:

- поля записи могут разделяться пробелами и запятой;
- если между полями записи присутствует слеш (/), то ввод прекращается;
- каждый оператор ввода (если не задан спецификатор ADVANCE = 'NO') выполняет ввод с начала новой записи. Например, при вводе

  ```
  read *, x, y, z
  ```

  можно обойтись одной записью, например:

  ```
  1.1  2.2  3.3
  ```

  тогда как при вводе

  ```
  read *, x
  read *, y
  read *, z
  ```

  уже потребуется 3 записи, например:

1.1
2.2
3.3

Причем если создать, например, в первой строке больше полей ввода:

1.1 4.4 5.5
2.2
3.3

то поля с символами 4.4 и 5.5 будут в последней версии ввода проигнорированы и по-прежнему после ввода: $x = 1.1$, $y = 2.2$, $z = 3.3$;

- если число элементов списка ввода больше числа полей записи, то для ввода недостающих значений оператор ввода перейдет к следующей записи;
- для ввода значения логической переменной достаточно набрать T или F.

Ошибки ввода возникают:

- если число элементов списка ввода больше числа доступных для чтения полей записи (т. е. если выполняется попытка чтения записи "конец файла" и за пределами файла);
- если размещенные на читаемом поле символы не могут быть приведены к типу соответствующего элемента списка ввода.

Пример:

```
read *, k
```

Ошибка ввода последует, если, например, ввести

```
k = 2
```

Правильный ввод:

```
2
```

1.8.2. Ввод из текстового файла

Ввод с клавиатуры даже сравнительно небольшого объема данных - достаточно утомительное занятие. Если, например, на этапе отладки вы многократно запускаете программу, то работа пойдет гораздо быстрее при вводе данных из файла.

Пусть надо определить вещественные переменные *x, y* и *z*, задав им при вводе значения 1.1, 2.2 и 3.3. Создадим файл a.txt в том же месте, откуда выполняется запуск программы, и занесем в него строку

```
1.1  2.2  3.3
```

Программа ввода из файла:

```
real x, y, z
open(2, file = 'a.txt')                ! 2 - номер устройства В/В
```

```
read(2, *) x, y, z                ! Ввод из файла a.txt
print *, x, y, z                  ! Вывод на экран
end
```

Оператор OPEN создает в программе устройство В/В и соединяет его с файлом a.txt. Далее в операторе READ вместо первой звездочки используется номер устройства, что обеспечивает ввод данных их файла, который с этим устройством связан. Правила ввода из текстового файла и с клавиатуры совпадают, поскольку клавиатура по своей сути является стандартным текстовым файлом.

1.8.3. Вывод на принтер

Принтер, как и клавиатуру, можно рассматривать как файл, который может быть подсоединен к созданному оператором OPEN устройству. Тогда программа вывода на принтер может выглядеть так:

```
real :: x = 1.1, y = 2.2, z = 3.3
open(3, file = 'prn')             ! Подсоединяем принтер к устройству 3
write(3, *) x, y, z               ! Вывод на принтер
write(*, *) x, y, z               ! Вывод на экран
end
```

1.9. Рекомендации по изучению Фортрана

Изучение языка программирования помимо чтения и разбора приведенного в книгах и пособиях материала включает и выполнение многочисленных, часто не связанных с практической деятельностью задач. Преимущественно такие задачи должны быть придуманы вами самостоятельно, поскольку постановка задачи помогает пониманию материала столь же эффективно, как и процесс ее анализа и решения.

К составлению и выполнению тестовых задач следует приступить начиная с первого дня изучения материала. Предположим, что прочитано несколько страниц 1-й главы. В соответствии с материалом вами набрана и запущена программа

```
program p1
real x, y, z
x = 1.1
y = 2.2
z = x + y
print *, 'z = ', z
end
```

Какие дополнительные шаги в рамках изучения языка могут быть предприняты? Возможно, следующие:

- выполним инициализацию переменных в операторе объявления:

```
real :: x = 1.1, y = 2.2
real z
print *, x, y                    ! Обязательно просматриваем результаты
```

или:

```
real x /1.1/, y /2.2/
real z
print *, x, y                    ! Обязательно просматриваем результаты
```

- дадим теперь начальные значения переменным оператором DATA:

```
real x, y, z
data x / 1.1 /, y / 2.2 /        ! или: data x, y / 1.1, 2.2 /
print *, x, y
```

- поместим в список вывода выражение и опустим заголовок программы:

```
real :: x = 1.1, y = 2.2
print *, x + y                   ! В списке вывода выражение
end
```

- запишем несколько операторов на одной строчке:

```
x = 1.1; y = 2.2; z = x + y      ! или: z = x + y;
```

- внесем ошибку в программу и посмотрим реакцию компилятора:

```
program p1
real :: x = 1.1, y = 2.2
print *, x + y
end program p11                  ! Ошибка: имя p11 не совпадает с именем p1
```

Составленные вами решения тестовых задач должны содержать достаточное число операторов вывода промежуточных результатов, которые позволят понять работу изучаемых элементов языка, убедиться в правильности вычислений или локализовать ошибку. Особенно внимательно следует наблюдать и проверять результаты выполняемого с клавиатуры и из файла ввода данных.

Уже на начальном этапе освоения материала следует не пожалеть времени для приобретения навыков ввода данных из текстового файла. Данные в текстовой файл могут быть занесены с клавиатуры в DS так же, как и в другом текстовом редакторе. В DS для создания нового файла используйте приведенные в разд. 1.2.1 и 1.2.2 сведения. Далее введите с клавиатуры данные, разделяя числа одним или несколькими пробелами, например:

```
1.2  -1.5  4.0  10   -3
```

Сохранить данные можно в любой существующей папке, однако на первых порах лучше размещать файл в папке, из которой выполняется запуск ваших учебных программ. Это освободит вас от необходимости задавать в программе путь в имени файла. Для записи файла на диск

используйте File - Save - в поле "Имя файла" задать имя файла, например a.txt - сохранить.

Теперь попробуем ввести данные из только что сформированного файла a.txt в одномерный массив *ar* из 10 элементов. Для этого мы должны открыть файл a.txt и поместить в список ввода оператора READ элементы массива, в которые выполняется ввод данных, например:

```
integer, parameter :: n = 10        ! Размер массива ar
real :: ar(n)                       ! Объявляем вещественный массив ar
character(50) :: fn = 'a.txt'       ! Задаем имя файла
ar = 0.0                            ! Теперь все элементы ar равны нулю
open(1, file = fn)                  ! Подсоединяем файл к устройству 1
! Ввод первых пяти элементов из файла a.txt
read(1, *) ar(1), ar(2), ar(3), ar(4), ar(5)
! Вывод первых пяти элементов массива ar на экран
print *, ar(1), ar(2), ar(3), ar(4), ar(5)
end
```

В списке ввода размещено 5 элементов массива. Что будет, если добавить в него еще один элемент, например *ar*(6)? Если ответ на вопрос не очевиден, то добавьте, запустите программу и объясните природу ошибки.

Использованные в примере списки В/В выглядят громоздко. Легко представить размер подобного списка при вводе, например, нескольких сотен элементов массива. В Фортране есть несколько способов задания компактных списков В/В. Например:

```
! Список ввода содержит все элементы массива ar
read(1, *) ar
! Циклический список ввода, содержащий пяти первых элементов массива ar
read(1, *) (ar(i), i = 1, 5)
! В списке ввода сечение массива ar из пяти первых его элементов
read(1, *) ar(1:5)
```

Таким же образом могут быть составлены и списки вывода.

Наиболее компактно выглядит первый список ввода в операторе READ(1, *) *ar*, но в нашем случае он нам не подходит. Почему?

В/В массива можно выполнить и в цикле

```
do i = 1, 5
  read(1, *) ar(i)                  ! В списке ввода один элемент массива ar
end do
```

Такой цикл эквивалентен последовательности 5 операторов ввода:

```
read(1, *) ar(1)                    ! В списке ввода один элемент массива ar
read(1, *) ar(2)
read(1, *) ar(3)
```

```
read(1, *) ar(4)
read(1, *) ar(5)
```

Однако такая последовательность, хотя в ней присутствует только 5 элементов массива, не может быть введена из созданного нами файла a.txt. Почему?

Вывод массива *ar* при помощи цикла

```
do i = 1, 5
  write(1, *) ar(i)                ! Вывод в файл a.txt
  write(*, *) ar(i)                ! Вывод на экран
end do
```

будет выполнен успешно. При этом, однако, выводимые данные будут размещены в столбик. Почему? Кстати, где расположатся выводимые в файл a.txt записи?

Добавим теперь в файл a.txt не менее пяти чисел. Если файл открыт в DS, то для перехода в окно с данными файла можно нажать Ctrl + F6 или выбрать окно с файлом, воспользовавшись пунктом меню Window. Пусть модифицированный файл выглядит так:

```
1.2  -1.5  4.0  10   -3
34.2 -55   79.1
90 100.2   -0.4
```

Теперь для ввода всех элементов массива можно применить оператор

```
read(1, *) ar
```

Кстати, почему можно размещать вводимые одним оператором READ данные на разных строчках файла?

Напишите теперь программу вывода первых девяти элементов массива *ar* на трех строках экрана по 3 числа массива в каждой строке.

Подобным образом следует анализировать и другие элементы Фортрана, сочетая чтение литературы, разбор приведенных в ней примеров с постановкой и решением учебных задач.

1.10. Обработка программы

Программист записывает программу в *исходном коде* (тексте). Программа может существовать в одном или нескольких файлах, называемых *исходными файлами*. Имена исходных файлов имеют расширения F90, FOR или F, например koda.f90. По умолчанию Фортран считает, что файлы с расширением F90 написаны в свободной форме, а с расширениями FOR и F - в фиксированной.

Далее выполняется компиляция программы, в результате которой исходный текст преобразовывается в *объектный код*. В процессе компиляции проверяется правильность составления программы и при

обнаружении синтаксических ошибок выдаются соответствующие сообщения. *Объектный код* - это запись программы в форме, которая может быть обработана аппаратными средствами. Такой код содержит точные инструкции о том, что компьютеру предстоит сделать. Отдельные компоненты программы могут быть откомпилированы раздельно. Часть компонентов может быть записана в библиотеку объектных файлов. Программа, преобразовывающая исходный код в объектный, называется *компилятором* или *транслятором*. Файлы с объектным кодом - объектные файлы - имеют расширение OBJ, например koda.obj.

На следующей стадии обработки выполняется сборка приложения. Часть объектных файлов может быть загружена из библиотек. При этом отдельные компоненты (главная программа, модули, подпрограммы, функции) связываются друг с другом, в результате чего образуется готовая к выполнению программа - *исполняемый файл*. Расширение таких файлов EXE. Программа, осуществляющая сборку, называется *компоновщиком* или *построителем*. На этапе генерации исполняемого кода также могут возникать ошибки, например вызов несуществующей подпрограммы.

В CVF и FPS подготовка исходного, объектного и исполняемого кодов может быть выполнена в специальной среде - Microsoft Developer Studio. Причем из одного проекта можно генерировать несколько реализаций. Например, на этапе разработки программы можно работать с реализацией, в которой отсутствует оптимизация исполняемого кода по его размеру и скорости выполнения (заданы опция компилятора /Od и опция компоновщика /OPT:NOREF). Отсутствие подобной оптимизации повышает скорость компиляции и компоновки. После завершения отладки можно создать рабочий проект, оптимизированный по размеру и скорости выполнения исполняемого файла, задав, например, при компиляции опцию /Oxp, а для компоновки - /OPT:REF. Можно задать и другие опции компилятора. Так, опция /G5 позволяет сгенерировать код, оптимально работающий на процессоре Intel Pentium.

По умолчанию при создании нового проекта в DS оказываются доступными две реализации: Debug и Release. В Debug активизирован отладочный режим. В Release исполняемый код оптимизируется по размеру и быстродействию. Ниже приведены задаваемые по умолчанию в FPS опции компилятора и компоновщика в реализациях Debug и Release при создании консоль-проекта.

Реализация Debug

Опции компилятора:

```
/Zi /I "Debug/" /c /nologo /Fo"Debug/" /Fd"Debug/koda.pdb"
```

Опции компоновщика:

```
kernel32.lib /nologo /subsystem:console /incremental:yes
/pdb:"Debug/koda.pdb" /debug /machine:I386 /out:"Debug/koda.exe"
```

Реализация Release

Опции компилятора:

```
/Ox /I "Release/" /c /nologo /Fo"Release/"
```

Опции компоновщика:

```
kernel32.lib /nologo /subsystem:console /incremental:no
/pdb:"Release/koda.pdb" /machine:I386 /out:"Release/koda.exe"
```

Замечание. Вопросы оптимального использования опций компилятора и построителя рассмотрены в [1].

2. Элементы программирования

2.1. Алгоритм и программа

Программа выполняет на ЭВМ некоторую последовательность действий, в результате чего должны получаться необходимые результаты. Для составления программы прежде необходимо уяснить суть задачи, а затем уже описать действия, после выполнения которых будут получены сформулированные в задаче цели. Иными словами, необходимо составить *алгоритм* решения задачи.

Рассмотрим простой пример. Пусть надо составить таблицу значений функции $y = x*\sin x$ на отрезке $[a, b]$ с шагом dx. Для решения задачи необходимо выполнить следующие действия:

1°. Начало.
2°. Ввести значения a и b границ отрезка и шаг dx.
3°. Задать x - начальную точку вычислений, приняв $x = a$.
4°. Пока $x \le b$, выполнять:
вычислить значение функции y в точке x: $y = x*\sin x$;
вывести значения x и y;
перейти к следующей точке отрезка: $x = x + dx$.
конец цикла 4:
5°. Конец.

Четвертый пункт алгоритма предусматривает повторное выполнение вычислений y для разных значений аргумента x. Такое повторное выполнение однотипных действий называется *циклом*. Приведенный цикл завершится, когда значение x превысит b - правую границу отрезка.

Для составления программы, выполняющей предусмотренные алгоритмом действия, надо перевести отдельные шаги алгоритма на язык программирования. Если буквально следовать приведенному алгоритму, то мы получим программу:

```
read *, a, b, dx                ! Выполняем 2-й шаг алгоритма
x = a                           ! Выполняем 3-й шаг алгоритма
do while(x <= b)                ! Выполняем 4-й шаг алгоритма
 y = x*sin(x)
 print *, x, y                  ! Вывод x и y
 x = x + dx
end do                          ! Конец цикла
end                             ! Завершаем программу
```

Однако, хотя программа записана правильно, работать с ней практически невозможно. Предположим, что вы все же запустили программу для вычислений. Тогда перед вами окажется черный экран, глядя на который вам придется догадаться, что нужно ввести 3 числа. Предположим, что вы догадались и, угадав затем порядок ввода, набрали 0,

1, 0.1. Тогда после нажатия на Enter перед вами появятся два столбика с цифрами и опять придется угадывать, что за числа в них расположены. Запустив эту программу через неделю, вы, конечно, уже ничего не вспомните.

Поэтому нам следует придать программе некоторые иные, не предусмотренные первоначальным алгоритмом свойства. Так, нужно создать диалог для ввода данных, нужно пояснить, каким объектам принадлежат выводимые на экран значения. Иными словами, нужно создать некоторый интерфейс между пользователем и программой. Нужно также проверить правильность ввода данных: левая граница должна быть меньше правой, а шаг *dx* должен быть больше нуля (в противном случае мы можем получить бесконечный цикл, например если $a < b$ и $dx \le 0$). Вводя подобные проверки, мы повышаем надежность программы. Можно предусмотреть и другие повышающие качество программы мероприятия.

Помимо добавлений таких рабочих характеристик полезно увеличить и требования к оформлению программы: дать программе имя, объявить типы применяемых в вычислениях переменных, привести исчерпывающий комментарий. Можно выполнить запись операторов, имен встроенных процедур и других элементов Фортрана прописными буквами, а запись введенных пользователем имен - строчными. При записи фрагментов программы, например управляющей конструкции DO WHILE ... END DO, следует использовать правило *рельефа*, состоящее в том, что операторы DO WHILE и END DO начинаются с одной позиции, а расположенные внутри этой конструкции операторы смещаются на одну-две позиции вправо по отношению к начальной позиции записи DO WHILE и END DO. Еще одно полезное правило: после запятой или иного разделителя в тексте программы следует проставлять *пробел*, т. е. поступать так же, как и при записи текста на родном языке. После ряда дополнений мы можем получить программу:

```
program txy                          ! Заголовок программы
 real a, b, dx, x, y                 ! Объявление имен и типов переменных
 real :: dxmin = 1.0e-4
 print *, 'Ввод границ отрезка и шага вычислений'
 print *, 'Левая граница: '          ! Выводим подсказку для пользователя
 read *, a                           ! Вводим с клавиатуры значение a и
 print *, 'Правая граница: '         ! нажимаем на Enter. Так же вводятся
 read *, b                           ! другие данные
 print *, 'Шаг вычислений: '
 read *, dx
 if(dx < dxmin) stop 'Ошибка при задании шага'
 x = a                               ! Выполняем 3-й шаг алгоритма
 do while(x <= b)                    ! Выполняем 4-й шаг алгоритма
   y = x*sin(x)                      ! При записи цикла используем правило рельефа
   print *, 'x = ', x, '   y = ', y
```

```
  x = x + dx
 end do
end program txy                    ! Завершаем программу txy
```

Замечание. CVF и FPS обладают специальной библиотекой процедур DIALOGM, предназначенных для создания диалоговых окон В/В данных. Технология создания диалогов средствами DIALOGM рассмотрена в [1].

Предположим, однако, что после ввода $a = 0$, $b = 1$ и $dx = 0.1$. Тогда для значений x, равных, например, 0.3, 0.4 и 0.5, будет выведен результат:

```
x =   3.000000E-01   y =   8.865607E-02
x =   4.000000E-01   y =   1.557673E-01
x =   5.000000E-01   y =   2.397128E-01
```

Каждая выводимая на экран строка является в нашем примере отдельной записью.

Результат вывода понятен, но не очень нагляден. Форму его представления можно улучшить, применив форматный вывод, т. е. задав некоторые правила преобразования выводимых данных. Такие правила задаются *дескрипторами преобразований* (ДП). Для вывода заключенной в кавычки последовательности символов используем дескриптор A, а вывод значения x выполним на поле длиной в 5 позиций, располагая после десятичной точки две цифры. Для этого нам понадобится дескриптор F5.2. При выводе y используем дескриптор F6.4. Тогда оператор вывода x и y примет вид:

```
print '(1x, a, f5.2, a, f6.4)', 'x = ', x, '   y = ', y
```

Результат для тех же значений x будет выглядеть уже более наглядно:

```
x =  .30   y =  .0887
x =  .40   y =  .1558
x =  .50   y =  .2397
```

Заметим сразу, что список ДП открывает дескриптор 1X, который означает задание одного пробела, предваряющего выводимый текст. В FPS при форматном выводе это необходимо, поскольку первый символ каждой записи не печатается и рассматривается как символ управления кареткой. В CVF такой интерпретации первого символа не выполняется и дескриптор 1X можно опустить.

Можно также улучшить и фрагмент программы, предназначенный для ввода данных. Если вы запустите программу, то обнаружите, что после вывода каждой подсказки курсор смещается на начало новой строки. Иными словами, выполняется переход на начало следующей записи. Такого перехода можно избежать, если использовать при выводе подсказки форматный вывод и применить в нем для вывода строки дескриптор A, а вслед за ним дескриптор $ или \ или спецификатор ADVANCE = 'NO'. Правда, последняя опция применима лишь в операторе вывода WRITE.

Выполняемый таким образом вывод называется *выводом без продвижения*. Например:

```
print '(1x, a, $)', 'Левая граница: '
print '(1x, a, \)', 'Левая граница: '
write(*, '(1x, a)', advance = 'no') 'Левая граница: '
```

Рассмотренный пример позволяет сделать, по крайней мере, 3 вывода.

С одной стороны, разработанный алгоритм:

- позволяет понять, какие данные являются входными, какие - результатом (т. е. выделить входные и выходные данные);
- описывает, какие действия должны быть выполнены программой для достижения результата;
- определяет порядок выполнения действий;
- устанавливает момент завершения вычислений.

Основой для программной реализации алгоритма являются управляющие конструкции, одной из которых является только что использованная конструкция DO WHILE ... END DO.

С другой стороны, очевидно, что для перевода алгоритма в программу необходимо обладать дополнительными, не связанными с алгоритмом знаниями. Например: как объявить типы данных, как создать приемлемый интерфейс между пользователем и программой, что такое запись, как выполнить форматный вывод, и что такое дескрипторы преобразований, и т. д.

И наконец, последний вывод: программист должен одинаково хорошо владеть и техникой составления алгоритмов, и техникой программирования, для освоения которой в современном Фортране, надо признать, требуется проделать большую работу.

2.2. Базовые структуры алгоритмов

Запись программы на языке программирования следует выполнять после разработки алгоритма. Имея алгоритм, вы знаете, как решать задачу, и во многом уже определяете контуры будущей программы. Здесь мы говорим лишь о контурах программы, поскольку реализация алгоритма в виде исходного кода может быть выполнена несколькими способами.

Для записи алгоритмов могут быть использованы *линейные схемы*, *блок-схемы* и *псевдокод*. Мы будем использовать линейные схемы, первый пример использования которой приведен в разд. 2.1. При их написании будем пользоваться правилом рельефа; необязательные элементы схем будем указывать в квадратных скобках, а знак вертикальной черты будем употреблять для обозначения "или".

Любой алгоритм может быть записан при помощи трех *базовых структур*:

- блока операторов и конструкций;

- ветвления;
- цикла.

2.2.1. Блок операторов и конструкций

Блок операторов и конструкций (БОК) - это выполнение одного или нескольких простых или сложных действий. БОК может содержать и ветвления и циклы, которые являются примерами сложных действий. Простым действием является, например, выполнение присваивания, В/В данных, вызов процедуры. *Конструкции* состоят из нескольких операторов и используются для выполнения управляющих действий, например циклов. Так, конструкция DO ... END DO состоит из двух операторов: DO и END DO.

2.2.2. Ветвление

Ветвление - выбор одного из возможных направлений выполнения алгоритма в зависимости от значения некоторых условий.

Различают ветвления четырех видов:

- если - то;
- если - то - иначе;
- если - то - иначе - если;
- выбор по ключу.

Здесь мы рассмотрим только два первых ветвления.

В ветвлениях "если - то" и "если - то - иначе" для записи условий используется *логическое выражение* (ЛВ), результатом которого может быть *истина* (И) или *ложь* (Л). Ветвления можно проиллюстрировать графически (рис. 2.1).

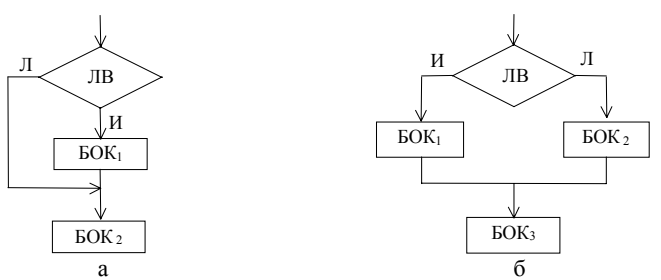


Рис. 2.1. Ветвления: а - ветвление "если - то"; б - ветвление "если - то - иначе"

Ветвление "если - то" работает так:

- вычисляется значение ЛВ;
- если оно истинно, то выполняется $БОК_1$;
- если оно ложно, то управление передается $БОК_2$.

Запись ветвления "если - то" в линейной схеме алгоритма:

```
X°.   Если истинно ЛВ, то [выполнить:]
        БОК1
      конец если [X°].
```

или, если в $БОК_1$ входит один оператор:

```
X°.   Если истинно ЛВ, то [выполнить:] оператор
```

На Фортране такое ветвление можно записать так:

```
IF(ЛВ) THEN
  БОК1
END IF
```

или так:

IF(ЛВ) *оператор*

Замечание. Оператор END IF можно записать и без пробела: ENDIF.

Ветвление "если - то - иначе" работает так:

- вычисляется значение ЛВ;
- если оно истинно, то выполняется $БОК_1$;
- если оно ложно, то выполняется $БОК_2$;
- далее управление передается $БОК_3$.

Запись ветвления "если - то - иначе" в линейной схеме алгоритма:

```
X°.   Если истинно ЛВ, то [выполнить:]
        БОК1
      иначе [выполнить:]
        БОК2
      конец если [X°].
```

Запись ветвления "если - то - иначе" на Фортране:

```
IF(ЛВ) THEN
  БОК1
ELSE
  БОК2
END IF
```

Для записи ЛВ используются логические операции и операции отношения. Также в ЛВ могут присутствовать арифметические и символьные операции. Приведем в табл. 2.1 некоторые логические операции и операции отношения в порядке убывания их приоритета. Обратите внимание, что операции отношения могут быть записаны в двух формах. В табл. 2.1 эти формы указаны одна под другой. Следует также подчеркнуть, что операция логического равенства записывается, если вы не используете форму .EQ., двумя знаками равенства (==). Пробелы в записи логических операций и операций отношения не допускаются, так, в случае операции "больше или равно" ошибочны записи .GE . и > =.

Таблица 2.1. Некоторые логические операции и операции отношения

Операции	*Запись на Фортране*	*Типы операций*
=, ≠, >, <, ≥, ≤	.EQ., .NE., .GT., .LT., .GE., .LE. ==, /=, >, <, >=, <=	Отношения
НЕ (отрицание)	.NOT.	Логическая
И	.AND.	"
ИЛИ	.OR.	"

Пример ветвления "если - то". Определить, какое из трех заданных чисел *ma*, *mb* и *mc* является наименьшим.

Алгоритм:

1°. Начало.
2°. Найти наименьшее из трех чисел и присвоить результат переменной *m*3.
3°. Если *ma* равно *m*3, то вывести сообщение "Число *ma*".
4°. Если *mb* равно *m*3, то вывести сообщение "Число *mb*".
5°. Если *mc* равно *m*3, то вывести сообщение "Число *mc*".
6°. Конец.

Данный алгоритм позволяет найти и вывести все числа, значения которых равны минимальному.

```
program fimin
 real :: ma = 5.3, mb = 7.6, mc = 5.3, m3
 m3 = min(ma, mb, mc)                    ! Вычисление минимума
 if(ma == m3) write(*, *) 'Число ma'
 if(mb == m3) write(*, *) 'Число mb'
 if(ma == m3) write(*, *) 'Число mc'
 write(*, *) 'Минимум равен ', m3
end program fimin
```

Результат:

```
Число ma
Число mc
Минимум равен    5.300000
```

2.2.3. Цикл

Цикл - повторное выполнение БОК, завершаемое при выполнении некоторых условий. Однократное выполнение БОК цикла называется *итерацией*. Операторы и конструкции БОК цикла также называются *телом цикла*.

Различают 3 вида циклов:

- цикл "с параметром";
- цикл "пока";
- цикл "до".

2.2.3.1. Цикл "с параметром"

В цикле *"с параметром p"* задаются начальное значение параметра p_s, конечное значение параметра p_e и шаг s - отличная от нуля величина, на которую изменяется значение параметра p после выполнения очередной итерации. Параметр p также называют переменной цикла, которая имеет целый тип. Параметры p_s, p_e и шаг s являются выражениями целого типа.

Графически цикл "с параметром" иллюстрирует рис. 2.2.

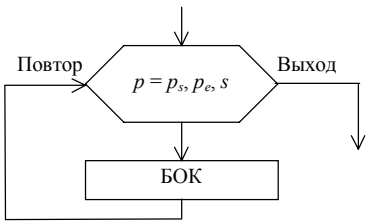


Рис. 2.2. Цикл с параметром

Цикл "с параметром" работает так (случай $s > 0$):

1°. Присвоить: $p = p_s$.
2°. Если $p \le p_e$, то перейти к п. 3°, иначе завершить цикл.
3°. Выполнить БОК.
4°. Присвоить: $p = p + s$ и перейти к п. 2° (повтор).

Когда $s < 0$, п. 2° выглядит так:

2°. Если $p \ge p_e$, то переход к п. 3°, иначе завершить цикл.

Замечания:

1. В цикле "с параметром" приведенные в пп. 1° и 4° операторы в тексте программы не присутствуют, но будут автоматически встроены компилятором в объектный код при компиляции программы.

2. В цикле "с параметром" запрещается в теле цикла менять значения переменной цикла p. Изменение параметров p_s, p_e и шага s в теле цикла не отразится на выполнении цикла: цикл будет выполняться с теми значениями параметров, какие они имели до начала первой итерации цикла.

Запись цикла "с параметром" в линейной схеме алгоритма:

X°. С параметром $p = p_s, p_e, s$ [выполнить:]
БОК
конец цикла [с параметром p] | [X°].

Наиболее часто цикл "с параметром" записывается так:

DO $p = p_s, p_e$ [, s]
БОК
END DO

При отсутствии шага *s* его значение устанавливается равным единице.

Замечание. Оператор END DO можно записать и без пробела: ENDDO.

Пример. Вычислить длину состоящей из *n* отрезков ломаной линии. Длины отрезков линии составляют последовательность a, $4a$, ..., n^2a.

Пусть *L* - искомая длина ломаной линии. Очевидно, если первоначально положить *L* = 0, то, выполнив *n* раз оператор

$$L = L + i^2 * a \quad (i = 1, 2, ..., n), \qquad (*)$$

где *i* - номер отрезка ломаной линии, мы получим искомый результат. Для *n*-кратного выполнения оператора (*) следует использовать цикл. Наиболее подходит для данной задачи цикл "с параметром", в котором в качестве параметра используется номер отрезка ломаной линии.

Алгоритм:

1°. Начало.
2°. Ввести значения *n* и *a*.
3°. Принять L = 0.0. ! L - длина ломаной линии
4°. С параметром *i* = 1, *n*, 1 выполнить: ! *i* - номер отрезка
    L = L + *i***2 * a
    конец цикла 4°.
5°. Вывод L.
6°. Конец.

```
program polen                          ! Программная реализация алгоритма
 integer i, n
 real a, L                             ! L - длина ломаной линии
 write(*, *) 'Введите a и n:'
 read(*, *) a, n
 L = 0.0
 do i = 1, n
  L = L + i**2 * a
 end do
 write(*, *) ' L = ', L
end program polen
```

Замечание. Скорость выполнения программы можно увеличить, если:

- вынести из цикла операцию умножения на переменную *a*, значение которой в цикле не изменяется;
- заменить операцию возведения в квадрат *i***2 на более быструю операцию умножения *i* * *i*.

Тогда фрагмент модифицированной программы будет таким:

```
L = 0.0
do i = 1, n
  L = L + i * i
end do
write(*, *) ' L = ', L * a
```

2.2.3.2. Циклы "пока" и "до"

Цикл *"пока"* выполняется до тех пор, пока "истинно" некоторое ЛВ. Причем проверка истинности ЛВ выполняется перед началом очередной итерации. Цикл *"до"* отличается от цикла "пока" тем, что проверка истинности ЛВ осуществляется после выполнения очередной итерации. В Фортране не существует цикла "до", но его можно реализовать в объявляющей бесконечный цикл конструкции DO ... END DO. Графическая интерпретация циклов "пока" и "до" приведена на рис. 2.3.

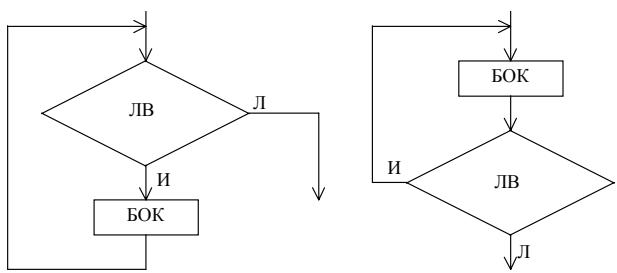


Рис. 2.3. Циклы "пока" и "до": а - цикл "пока"; б - цикл "до"

Замечание. При работе с циклами "пока" и "до" надо следить, чтобы ЛВ обязательно рано или поздно приняло значение *ложь*. Иначе произойдет зацикливание - "бесконечное" выполнение операторов цикла.

Запись циклов "пока" и "до" в линейной схеме алгоритма и на Фортране:

Цикл *"пока"*:

```
X°.  Пока истинно ЛВ [, выполнять:]
       БОК
     конец цикла X°.
```

```
DO WHILE(ЛВ)
  БОК
END DO
```

Цикл *"до"*:

```
X°.  Выполнять:
       БОК
       если ложно ЛВ, то выход из цикла
     конец цикла X°.
```

```
DO
  БОК
  IF(.NOT. ЛВ) EXIT
END DO
```

2.2.4. Прерывание цикла. Объединение условий

Выйти из цикла и передать управление на первый следующий за циклом выполняемый оператор можно, применив оператор EXIT. Если нужно пропустить часть операторов цикла и перейти к следующей итерации, то нужно использовать оператор CYCLE. При этом управление передается операторам DO или DO WHILE. Операторы EXIT и CYCLE отдельно не применяются, а встраиваются в конструкции IF.

Пример. Вычислить число положительных и отрицательных элементов одномерного массива *a* из *n* элементов, заканчивая вычисления, если число нулевых элементов массива превысит *k*.

```
program pn
integer, parameter :: n = 10
integer :: a(n) = (/ 1, -2, 0, 3, -4, 5, -6, 7, 0, 9 /)
integer :: k = 3, pos = 0, ze = 0, i, va
do i = 1, n                          ! i - номер элемента массива a
 va = a(i)
 if(va == 0) then
  ze = ze + 1                        ! ze - число нулевых элементов массива
  if(ze > k) then  exit              ! Выход из цикла
  else
   cycle                             ! Переход на начало цикла
  end if
 end if                              ! pos - число положительных элементов массива
 if(va > 0) pos = pos + 1
end do                               ! Число отрицательных элементов: n - ze - pos
if(ze > k) stop 'Число нулевых элементов больше нормы'
write(*, *) 'pos = ', pos, '  neg = ', n - ze - pos
end program pn
```

Замечание. Использование переменной *va* позволяет сократить число обращений к массиву *a* и тем самым повысить быстродействие программы.

В данной задаче для завершения цикла можно использовать широко применяемый в программировании метод *объединения условий*. Цикл должен продолжаться, пока истинны два условия: $i \le n$ и $ze \le k$. При нарушении одного из них цикл должен быть прекращен. Используем объединение условий в цикле "пока".

Алгоритм:

1°. Начало.
2°. Задать значения *n*, *a* и *k*.
3°. Принять:
   pos = 0 ! *pos* - число положительных элементов массива *a*
   ze = 0 ! *ze* - число равных нулю элементов массива *a*
   i = 1 ! *i* - текущий номер элемента массива *a*
4°. Пока $i \le n$ и $ze \le k$, выполнить:

```
      va = a(i)
      Если va = 0, то
        ze = ze + 1
      иначе, если va > 0, то
        pos = pos + 1
      конец если.
    конец цикла 4°.
5°. neg = n - ze - pos          ! neg - число отрицательных элементов массива a
6°. Вывод pos и neg.
7°. Конец.
```

```
program pnw             ! Программная реализация алгоритма
integer, parameter :: n = 10
integer :: a(n) = (/ 1, -2, 0, 3, -4, 5, -6, 7, 0, 9 /)
integer :: k = 3, pos = 0, ze = 0, i, va
i = 1                   ! Начинаем вычисления с первого элемента массива
do while(i <= n .and. ze <= k)
 va = a(i)
 if(va == 0) then
  ze = ze + 1           ! ze - число равных нулю элементов массива
 else if(va > 0) then
  pos = pos + 1         ! pos - число положительных элементов массива
 end if
 i = i + 1
end do                  ! Число отрицательных элементов: n - ze - pos
if(ze > k) stop 'Число нулевых элементов больше нормы'
write(*, *) 'pos = ', pos, '  neg = ', n - ze - pos
end program pnw
```

Идея объединения условий может быть реализована при помощи *флажка*, например:

```
...
logical fl              ! Флажок - переменная логического типа
i = 1
fl = i <= n .and. ze <= k   ! Начальное значение флажка fl
do while(fl)            ! Пока значение fl есть истина, цикл выполняется
 ...
  i = i + 1
  fl = i <= n .and. ze <= k   ! Новое значение флажка
end do
```

2.3. Программирование "сверху вниз"

Разработка алгоритмов и программ осуществляется, как правило, по принципу "сверху вниз".

Суть такого подхода состоит в разбиении исходной задачи на ряд более простых задач - фрагментов и последующей работе с полученными фрагментами.

При разбиении задачи на фрагменты надо придерживаться следующей схемы:

1) проанализировать задачу и выделить в ней фрагменты;
2) отобразить процесс разбиения в виде блок-схемы или линейной схемы и пронумеровать в ней фрагменты;
3) установить между выделенными фрагментами связи: для каждого фрагмента определить, какие данные он получает (входные данные) и какие данные возвращает (выходные данные). Связи между фрагментами называются *интерфейсом*;
4) рассмотреть далее каждый фрагмент самостоятельно; разработать для него алгоритм и записать его либо в виде линейной схемы, либо в виде блок-схемы. При необходимости подвергнуть фрагмент разбиению на более мелкие фрагменты. Такое разбиение продолжать до тех пор, пока не будут получены фрагменты, программирование которых не составляет особых затруднений;
5) оформить выделенные фрагменты в виде программных компонентов или БОК.

При таком подходе программу можно рассматривать как совокупность фрагментов, которые, принимая некоторые данные, вырабатывают результат и передают его следующему фрагменту.

Составляемые для фрагментов линейные схемы сопровождаются заголовком, описанием интерфейса (состава входных и выходных данных).

В Фортране для реализации фрагмента можно использовать программные единицы: *главную программу*, *модули*, *подпрограммы* и *функции*.

Подпрограммы и функции называются *процедурами* и могут быть *внешними*, *модульными* и *внутренними*.

Модули и внешние процедуры являются самостоятельными *программными единицами*, доступ к которым может быть выполнен из разных программ.

Подробное рассмотрение проблем разработки программных компонентов мы отложим до гл. 8. Здесь же проиллюстрируем методы программирования "сверху вниз".

2.3.1. Использование функций

Фрагмент алгоритма, как правило, оформляется в виде функции, если в результате выполненных в нем вычислений возвращается единственный скаляр или массив.

Пример. В каком из трех одномерных массивов $a(1{:}10)$, $b(1{:}15)$ и $c(1{:}20)$ первый отрицательный элемент имеет наименьшее значение.

Алгоритм:

1°. Ввести массивы *a*, *b* и *c*.
2°. Найти *ma* - значение первого отрицательного элемента в массиве *a*.
3°. Найти *mb* - значение первого отрицательного элемента в массиве *b*.
4°. Найти *mc* - значение первого отрицательного элемента в массиве *c*.
! Значения *ma*, *mb* или *mc* равны нулю, если в соответствующем массиве
! нет отрицательных элементов
5°. Если *ma* + *mb* + *mc* = 0, то
Вывести сообщение: "В массивах нет отрицательных элементов"
иначе
Проанализировать значения *ma*, *mb* и *mc* и вывести имя массива, в котором первый отрицательный элемент имеет наименьшее значение.
! Алгоритм этого фрагмента приведен в разд. 2.2.2
конец если 5:
6°. Конец.

Фрагменты 2°, 3° и 4° содержат одну и ту же решаемую для разных массивов задачу. В соответствии с методом программирования "сверху вниз" опишем интерфейс, т. е. входные и выходные данные фрагмента, а затем алгоритм его реализации.

Интерфейс фрагмента 2° (3°, 4°):

Входные данные. Одномерный массив и число его элементов. Используем внутри фрагмента для массива имя *d*, а для числа элементов массива - имя *n*.

Выходные данные. Значение первого отрицательного элемента массива *d* или 0, если в массиве *d* нет отрицательных элементов. Для результата используем имя *md*.

Алгоритм поиска первого отрицательного элемента массива:

1°. *i* = 1 ! *i* - номер элемента массива *d*
2°. *md* = *d*(*i*) ! Подготовка к циклу
3°. Пока *md* ≥ 0 и *i* < *n*, выполнять:
i = *i* + 1
md = *d*(*i*)
конец цикла 3°.
4°. Если *md* ≥ 0, то *md* = 0 ! Возвращаем нуль, если отрицательный
5°. Возврат. ! элемент не найден

Фрагмент возвращает одно значение (*md*), поэтому его можно реализовать в виде функции. В Фортране переменная, в которую заносится возвращаемый функцией результат, называется *результирующей* и ее тип и имя (если не задано предложение RESULT) совпадают с именем и типом функции. В нашем случае функция будет иметь имя *md*.

```
program nera                    ! Запись приведенных алгоритмов на Фортране
integer, parameter :: na = 10, nb = 15, nc = 20
integer a(na), b(nb), c(nc)
integer ma, mb, mc, m3          ! Поскольку функция md оформлена как
```

```
integer md                          ! внешняя, то необходимо объявить ее тип
<Ввод массивов a, b, и c>
ma = md(a, na)                      ! Передача входных данных (массива и числа
mb = md(b, nb)                      ! его элементов) в функцию md выполняется
mc = md(c, nc)                      ! через ее параметры
if(ma + mb + mc == 0) then
 print *, 'В массивах a, b и c нет отрицательных элементов'
else
 m3 = min(ma, mb, mc)
 if(ma == m3) print *, 'В массиве a'
 if(mb == m3) print *, 'В массиве b'
 if(mc == m3) print *, 'В массиве c'
end if
end program nera
```

```
function md(d, n)                   ! Заголовок функции
 integer md                         ! Результирующая переменная
 integer n, d(n), i                 ! Функция возвращает первый отрицательный элемент
 i = 1                              ! массива d или 0 при отсутствии таковых
 md = d(i)
 do while(md >= 0 .and. i < n)
   i = i + 1
   md = d(i)
 end do
 if(md > 0) md = 0                  ! или: md = min(md, 0)
end function md
```

2.3.2. Использование подпрограмм

Если фрагмент алгоритма возвращает более одного скаляра и/или массива, то такой фрагмент, как правило, оформляется в виде подпрограммы. Передача входных данных в подпрограмму и возвращаемых из нее величин выполняется через ее параметры.

Выполним подсчет числа положительных (*pos*), отрицательных (*neg*) и равных нулю (*ze*) элементов массива *a* из примера разд. 2.2.4 в подпрограмме *vapeg*. В нее мы должны передать массив *a*, и затем получить из нее искомые значения: *pos*, *neg* и *ze*:

```
program pns
<Объявление данных>                 ! См. текст программы pn разд. 2.2.4
call vapeg(a, n, pos, neg, ze)      ! Вызов внешней подпрограммы vapeg
if(ze > k) stop 'Число нулевых элементов больше нормы'
write(*, *) 'pos = ', pos, '  neg = ', neg
end program pns
```

```
subroutine vapeg(a, n, pos, neg, ze)
 integer :: n, a(n), pos, neg, ze, i, va
 pos = 0; neg = 0; ze = 0           ! Подготовка к вычислениям
```

```
<Вычисление pos, neg и ze>          ! См. разд. 2.2.4
end subroutine vapeg
```

2.3.3. Использование модулей

Модули могут объединять в себе данные и процедуры, которые выполняют обработку объявленных в модуле данных. Программная единица, в которой присутствует оператор USE *имя-модуля*, получает доступ к не имеющим атрибута PRIVATE данным и процедурам модуля.

Рассмотрим пример использования модуля для записи фрагмента алгоритма. Вернемся для этого к задаче разд. 2.1. В ней можно выделить два фрагмента:

1°. Начало.
2°. Ввести и проверить введенные значения границ отрезка *a*, *b* и шага *dx*.
3°. Если введенные данные не содержат ошибок, то
    Выполнить начиная с точки $x = a$ до точки *b* с шагом *dx* вычисление *y*
    и вывод значений *x* и *y*.
    конец если 3°.
4°. Конец.

Записи линейных схем фрагментов 2° и 3° легко выполнить по приведенному в разд. 2.1 алгоритму.

Фрагмент 2° хорошо вписывается в концепцию модуля: он содержит данные и некоторый код, выполняющий обработку данных, часть из которых затем будет использована во втором фрагменте. Реализацию третьего фрагмента, как и ранее, выполним в главной программе.

```
module ched                         ! Модуль ввода и обработки данных
 real a, b, dx                      ! Объявление данных модуля
 real, private :: dxmin = 1.0e-4
contains                            ! Далее следует модульная функция
 function moched( )                 ! ввода и обработки данных
  logical moched                    ! Тип модульной функции
  print *, 'Ввод границ отрезка и шага вычислений'
  print '(1x, a, $)', 'Левая граница: '
  read *, a
  print '(1x, a, $)', 'Правая граница: '
  read *, b
  print '(1x, a, $)', 'Шаг вычислений: '
  read *, dx
  moched = .false.                  ! Результирующая переменная moched равна
  if(dx < dxmin) then               ! .FALSE., если есть ошибки в данных
   print *, 'Ошибка при задании шага'
  else if(a >= b) then
   print *, 'Ошибка при задании границ отрезка'
  else
   moched = .true.                  ! Если нет ошибок в данных
```

```
  end if
 end function
end module
program txy                          ! Заголовок главной программы
use ched                             ! Ссылка на модуль CHED
real x, y                            ! Объявление данных главной программы
if(moched( )) then                   ! Вызов модульной функции
 x = a                               ! Выполняем вычисления, если введенные
 do while(x <= b)                    ! данные не содержат ошибок
   y = x * sin(x)
   print '(1x, a, f5.2, a, f6.4)', 'x = ', x, '   y = ', y
   x = x + dx
 end do
end if
end program txy
```

2.4. Этапы проектирования программ

Рассмотренный выше порядок создания программы включает этапы составления общей схемы решения задачи, выделения фрагментов и их интерфейсов (входных и выходных данных), разработки алгоритмов для фрагментов и последующего их кодирования. Если теперь их дополнить этапом тестирования и отладки, то получится схема, вполне пригодная для решения простых задач. Однако жизненный цикл крупных программ несколько шире и состоит из этапов:

1. Разработка спецификации.
2. Проектирование программы.
3. Запись программы на языке программирования (кодирование).
4. Отладка и тестирование программы.
5. Доработка программы.
6. Производство окончательного программного продукта.
7. Документирование.
8. Поддержка программы в процессе эксплуатации.

Спецификация содержит постановку задачи, анализ этой задачи и подробное описание действий, которые должна выполнять программа. В спецификации отражаются:

- состав входных, выходных и промежуточных данных;
- какие входные данные являются корректными и какие ошибочными;
- кто является пользователем программы и каким должен быть интерфейс;
- какие ошибки должны выявляться и какие сообщения должны выдаваться пользователю;
- какие ограничения имеет программа (например, программа размещения элементов печатной платы может иметь ограничение по числу размещаемых элементов);

- все особые ситуации, которые требуют специального рассмотрения;
- какая документация должна быть подготовлена;
- перспективы развития программы.

На этапе *проектирования* создается *структура* программы и для каждого фрагмента выбираются известные или разрабатываются новые *алгоритмы*. Последние должны быть подвергнуты тщательным исследованиям на предмет их результативности, т. е. способности алгоритма получать требуемые результаты, и эффективности - способности алгоритма получать нужные результаты за приемлемое время.

Параллельно с разработкой алгоритмов решаются вопросы *организации данных*, т. е. выделяются данные стандартных типов и способы их представления (скаляр или массив), а также разрабатываются новые структуры данных и определяется круг используемых с этими структурами операций. Подходы к решению этих задач во многом зависят от используемого языка программирования, который может быть, например, объектно-ориентированным или модульным. Современный Фортран поддерживает концепции как процедурного, так и модульного программирования (разд. 8.1).

Для каждого фрагмента на этом этапе также создаются полные *спецификации* по приведенной выше схеме.

Кодирование после разработки проекта программы и необходимых спецификаций является достаточно простой задачей. Когда же первые два этапа в явном виде не присутствуют, то неявно они выносятся на этап кодирования, со всеми вытекающими отсюда последствиями. Впрочем, для сложных задач игнорирование обозначенных выше этапов разработки программы недопустимо.

Тестирование - это запуск программы или отдельного фрагмента с целью выявления в них ошибок. *Отладка* - процесс локализации и исправления ошибок. В результате тестирования устанавливается, соответствуют или нет разработанные фрагменты и состоящая из них программа сформулированным в спецификациях требованиям. Методы тестирования и отладки рассмотрены, например, в [10].

Для тестирования фрагмента (программы) создаются специальные *тестовые наборы* входных данных, для которых до запуска фрагмента (программы) вычисляются *ожидаемые* результаты. Запуск фрагмента выполняется из специально созданной вспомогательной программы, называемой *драйвером*. Если фрагмент, в свою очередь, вызывает другие фрагменты, работоспособность которых пока еще не проверена, то эти фрагменты заменяются специальными простыми программами, которые имеют тот же интерфейс, что и заменяемые фрагменты, и имитируют их деятельность. Такие программы называются *заглушками*.

Тестирование может начинаться с фрагментов низшего уровня. Тогда нам понадобятся только драйверы, поскольку фрагменты более высокого уровня будут вызывать уже проверенные фрагменты. Такая стратегия тестирования называется *восходящей*. При *нисходящей* стратегии тестирование начинается с фрагментов высшего уровня. В этом случае понадобятся только заглушки. Обычно при тестировании восходящая и нисходящая стратегии используются совместно.

Проделанная на предшествующих этапах работа, как правило, предоставляет разработчикам достаточный материал, позволяющий сделать выводы о рабочих характеристиках программы и сформулировать предложения по улучшению программы и ее отдельных показателей. Однако не всегда все эти предложения сразу же реализуются и программа с определенными изъянами выходит в свет в качестве *программного продукта*. Впрочем, если некоторые характеристики серьезно ухудшают качество программы, то придется выполнить ее *доработку*.

Поддержка программы в процессе эксплуатации имеет целью устранение выявленных пользователями ошибок и адаптацию программного продукта к условиям его эксплуатации. Помимо этого, накапливается материал, необходимый для последующего развития и создания новой версии программы.

2.5. Правила записи исходного кода

Программисты, как правило, со временем вырабатывают свой стиль записи исходного кода, позволяющий им при повторном обращении к программе быстро вспоминать, что она делает и как работает, и при необходимости быстро вносить изменения в программу. Иными словами, программист умеет писать хорошо читаемый и легко изменяемый код. В ряде случаев необходимо, чтобы этот код легко могли бы прочесть и изменить другие программисты, например сопровождающие программу. Итак, за счет каких приемов удается записать хорошо читаемый и легко изменяемый код? Вот некоторые из них:

- программная единица должна содержать достаточный комментарий, позволяющий определить ее назначение, состав входных и выходных данных и выполняемые ей действия. Комментарий, однако, не должен мешать чтению операторов Фортрана;
- комментарий должен пояснять смысл используемых объектов данных;
- все используемые в программной единице данные должны быть явно объявлены. Это правило будет легче выполнить, если ввести в программную единицу оператор IMPLICIT NONE;
- операторы объявления следует группировать по типам;
- используемые для объектов данных и процедур имена должны напоминать их смысл и указывать на используемый тип. Например, имя

g может быть использовано для обозначения ускорения свободного падения, имя *iVectorSize* подойдет для указания длины вектора. Понятен смысл и имени *pi*;

- атрибуты объектов данных следует объявлять в операторах объявления типа, например так:

```
integer, parameter :: n = 20, m = 10      ! Размерности матрицы a
```

а не так:

```
integer n, m                              ! Этот способ хуже
parameter  (n = 20, m = 10)
```

- задание размерностей статических массивов лучше выполнять в виде именованных констант. В случае изменения размерности потребуется изменить лишь значение соответствующей константы:

```
integer, parameter :: m = 10, n = 20
real a(m, n)
```

- длину символьной именованной константы лучше задавать в виде звездочки, например:

```
character(*), parameter :: date = '01.01.2000'
```

- при записи управляющих конструкций следует использовать *правило рельефа*, состоящее в том, что расположенные внутри конструкции операторы записываются правее образующих эту конструкцию операторов, например операторов IF-THEN-ELSE. Это же правило распространяется на запись определений производных типов и процедур;
- при записи операторов и выражений следует использовать пробелы, например до и после оператора присваивания или логической операции. Не забывайте ставить пробелы и после запятых, например в конструкторе массива. Однако в длинных выражениях пробелы между знаками операций могут быть опущены;
- при создании вложенных конструкций им следует давать имена;
- операторы FORMAT группируются в одном месте, как правило вверху или внизу программной единицы;
- размещенные в одном файле программные единицы должны разделяться пустыми строками;
- программные единицы следует располагать в файле в алфавитном порядке их имен;
- однократно используемые в программе процедуры лучше оформлять как внутренние, следующие после оператора CONTAINS;
- внутренняя функция лучше операторной;
- формальный параметр - массив следует оформлять как массив, перенимающий форму, или как массив заданной формы;

- внутренние массивы и строки процедур следует оформлять как автоматические объекты;
- не использовать оператор GOTO;
- не применять ассоциирования памяти;
- отказаться от использования нерекомендуемых и устаревших средств Фортрана (прил. 2).

Большинство из приведенных правил рассмотрены в пособии и проиллюстрированы примерами.

3. Организация данных

Для каждого применяемого внутри программной единицы объекта данных должны быть определены тип, диапазон изменения (в случае числового типа), а также форма представления: скаляр или массив. Данные также должны быть разделены на изменяемые - переменные - и не подлежащие изменению - константы.

Помимо переменных и констант к объектам данных можно отнести и *функции*, поскольку они так же, как и переменные и константы, обладают типом и используются в качестве операндов выражений.

Термины *переменная* и *константа* распространяются на скаляры, массивы и их подобъекты, например элементы массивов, компоненты структур, сечения массивов, подстроки.

Цель настоящей главы - рассмотрение типов данных Фортрана, их свойств, способов объявления объектов данных разных типов, способов задания начальных значений переменных и других, связанных с организацией данных вопросов. Диапазон рассмотрения ограничен скалярами. Массивы, как более сложные, играющие исключительно важную роль в Фортране объекты данных, рассмотрены отдельно.

Замечание. При описании операторов Фортрана их необязательные элементы заключаются в квадратные скобки. Символ вертикальной черты (|) используется в описании оператора для обозначения "или".

3.1. Типы данных

Типы данных разделяются на встроенные и производные, создаваемые пользователем (разд. 3.9).

Встроенные типы данных:

Целый - INTEGER, BYTE, INTEGER(1), INTEGER(2), INTEGER(4).

Вещественный - REAL, REAL(4), REAL(8), DOUBLE PRECISION.

Комплексный - COMPLEX, COMPLEX(4), COMPLEX(8), DOUBLE COMPLEX.

Логический - LOGICAL, LOGICAL(1), LOGICAL(2), LOGICAL(4). Объект данных логического типа может принимать значения .TRUE. (*истина*) или .FALSE. (*ложь*).

Символьный - CHARACTER(*n*), где *n* - длина символьной строки ($1 \le n \le 32767$).

В Фортране каждый встроенный тип данных характеризуется параметром разновидности (KIND). Для числовых типов данных этот параметр описывает точность и диапазон изменения. В настоящее время для символьного типа данных существует только одна разновидность (KIND = 1).

Каждый встроенный тип данных имеет стандартную, задаваемую по умолчанию разновидность. Встроенный тип с задаваемой по умолчанию разновидностью называется *стандартным типом данных*.

Стандартные типы данных:
Целый - INTEGER.
Вещественный - REAL.
Комплексный - COMPLEX.
Логический - LOGICAL.
Символьный - CHARACTER.

В табл. 3.1 приведены разновидности встроенных типов данных. В графе "Число байт" указано количество байт, отводимых под объект заданного типа. При обозначении типа данных использован введенный в стандарте Фортран 90 синтаксис.

Таблица 3.1. Разновидности встроенных типов данных

Типы	*Разновидность*	*Число байт*	*Примечание*
		Целый тип	
BYTE	1	1	То же, что и INTEGER(1)
INTEGER(1)	1	1	
INTEGER(2)	2	2	
INTEGER(4)	4	4	
INTEGER	4	4	То же, что и INTEGER(4)
		Вещественный тип	
REAL(4)	4	4	
REAL	4	4	То же, что и REAL(4)
REAL(8)	8	8	
DOUBLE PRECISION	8	8	То же, что и REAL(8)
		Комплексный тип	
COMPLEX(4)	4	8	4 байта под действительную и столько же под мнимую часть
COMPLEX	4	8	То же, что и COMPLEX(4)
COMPLEX(8)	8	16	8 байт под действительную и столько же под мнимую часть
DOUBLE COMPLEX	8	16	То же, что и COMPLEX(8)

Логический тип			
LOGICAL(1)	1	1	Байт, содержащий либо 0 - .FALSE., либо 1 - .TRUE.
LOGICAL(2)	2	2	Первый (старший) байт содержит значение LOGICAL(1), второй - *null*
LOGICAL(4)	4	4	Первый байт содержит значение LOGICAL(1), остальные - *null*
LOGICAL	4	4	То же, что и LOGICAL(4)
Символьный тип			
CHARACTER или CHARACTER(1)	1	1	Единичный символ
CHARACTER(*n*)	1	*n*	*n* - длина строки в байтах

Замечания:

1. Каждый числовой тип данных содержит 0, который не имеет знака.

2. Все приведенные в табл. 3.1 типы данных были доступны и в Фортране 77. Правда, синтаксис определения типа был иным, например:

Фортран 90	*Фортран 77*
INTEGER(1)	INTEGER*1
INTEGER	INTEGER
COMPLEX(4)	COMPLEX*8
COMPLEX(8)	COMPLEX*16

В Фортране 90 после описания встроенного типа данных в скобках указывается значение параметра разновидности, а в Фортране 77 после звездочки следует число отводимых под тип байт. С целью преемственности можно использовать при задании типов данных синтаксис Фортрана 77.

Помимо встроенных типов можно задать и производные типы данных (структуры), которые создаются комбинаций данных, встроенных и ранее введенных производных типов. Такие типы данных вводятся оператором TYPE ... END TYPE.

Фортран включает большое число встроенных числовых справочных и преобразовывающих функций (разд. 6.11-6.13), позволяющих получать информацию о свойствах данных различных типов. Так, наибольшее положительное число для целого и вещественного типов определяется функцией HUGE, а наименьшее положительное число вещественного типа - функцией TINY.

3.2. Операторы объявления типов данных

3.2.1. Объявление данных целого типа

Оператор INTEGER объявляет переменные, константы, функции целого типа. Объекты данных целого типа могут быть заданы как INTEGER, INTEGER(1), INTEGER(2) или INTEGER(4). Можно использовать и принятый в Фортране 77 синтаксис: INTEGER*1, INTEGER*2 или INTEGER*4. Напомним, что указанные в скобках значения задают разновидность типа, значения после звездочки - число отводимых под тип байт. Задаваемую по умолчанию разновидность стандартного целого типа данных INTEGER можно изменить, используя опцию компилятора /4I2 или директиву $INTEGER:2.

Синтаксис оператора INTEGER:

INTEGER [[([KIND =] *kind-value*)][, *attrs*] ::] *entity-list*

kind-value - значение параметра разновидности KIND. В качестве *kind-value* может быть использована ранее определенная именованная константа.

attrs - один или более атрибутов, описывающих представленные в *entity-list* объекты данных. Если хотя бы один атрибут указан, то должен быть использован разделитель :: . Возможные атрибуты: ALLOCATABLE, AUNOMATIC, DIMENSION(*dim*), EXTERNAL, INTENT, INTRINSIC, OPTIONAL, PARAMETER, POINTER, PRIVATE, PUBLIC, SAVE, STATIC, TARGET и VOLATILE (последний атрибут применим только в CVF). Задание атрибутов может быть также выполнено и отдельным оператором, имя которого совпадает с именем атрибута. Атрибуты определяют дополнительные свойства данных и будут вводиться по мере изложения материала. Если задан атрибут PARAMETER, то необходимо инициализирующее выражение, например:

```
integer(4), parameter :: m= 10, n = 20
```

entity-list - разделенный запятыми список имен объектов данных (переменных, констант, а также внешних, внутренних, операторных и встроенных функций), содержащий необязательные инициализирующие значения переменных выражения. При этом инициализация может быть выполнена двумя способами:

```
integer :: a = 2, b = 4              ! Наличие разделителя :: обязательно
```

или

```
integer a /2/, b /4/                 ! Разделитель :: может отсутствовать
```

Если параметр KIND отсутствует, то применяемое по умолчанию значение разновидности равно четырем (если не использована опция компилятора /4I2 или директива $INTEGER:2). Значение параметра разновидности можно узнать, применив встроенную справочную функцию KIND (разд. 6.10).

Диапазон изменения значений целых типов:

BYTE	то же, что и INTEGER(1)
INTEGER(1)	от -128 до +127
INTEGER(2)	от -32,768 до +32,767
INTEGER(4)	от -2,147,483,648 до +2,147,483,647
INTEGER	то же, что и INTEGER(4)

Пример:

```
integer day, hour                          ! Объявление без атрибутов
integer(2) k/5/, limit/45/                 ! Объявление и инициализация
byte vmin = -1                             ! То же, что и INTEGER(1)
                                           ! Объявления с атрибутами
integer, allocatable, dimension(:) :: days, hours
integer(kind = 2), target :: kt = 2        ! Объявление и инициализация
integer(2), pointer :: kp
integer(1), dimension(7) ::  val = 2       ! Объявление и инициализация
allocate(days(5), hours(24))               ! Размещение массивов
days = (/ 1, 3, 5, 7, 9/ )                 ! Генерация значений массивов
hours = (/ (i, i = 1, 24) /)               ! при помощи конструктора массива
day = days(5)
hour = hours(10)
kp => kt                                   ! Присоединение ссылки к адресату
print *, day, hour
print *, kt, kp
print *, vmin, val(5), k*limit, kind(val), range(kp)
end
```

Результат:

```
   9        10
   2         2
  -1         2       225         1         4
```

Замечание. Пользуясь синтаксисом Фортрана 77, первые 3 оператора можно объединить, указав размер типа в байтах после имени переменной:

```
integer day*4, hour*4, k*2 /5/, limit*2 /45/, vmin*1 /-1/
```

Число, стоящее после звездочки (*), указывает на количество байт, отводимых под переменную заданного типа. Такой способ объявления данных возможен и с другими встроенными типами.

Целая величина может быть в ряде случаев использована там, где ожидается логическое значение (в операторах и конструкциях IF и DO WHILE). При этом любое отличное от нуля целое интерпретируется как *истина* (.TRUE.), а равное нулю - как *ложь* (.FALSE.), например:

```
integer(4) :: i, a(3) = (/ 1, -1, 0 /)
do i = 1, 3
 if(a(i)) then
   write(*, *) 'True'
```

```
 else
   write(*, *) 'False'
 end if
end do
```

Результат:

```
True
True
False
```

Также целая величина может быть присвоена логической переменной:

```
logical fl1, fl2
fl1 = 5; fl2 = 0
print *, fl1, fl2                    ! T  F
```

Замечание. Смешение логических и целых величин недопустимо, если используется опция компилятора /4Ys или директива $STRICT, при которой все расширения по отношению к стандарту Фортран 90 воспринимаются как ошибки.

3.2.2. Объявление данных вещественного типа

Синтаксис оператора объявления объектов вещественного типа аналогичен синтаксису оператора INTEGER:

REAL [[([KIND =] *kind-value*)] [, *attrs*] ::] *entity-list*

Параметр KIND может принимать значения 4 и 8. Первое значение используется для объявления объектов данных одинарной точности, а второе - для объектов двойной точности. Параметр разновидности может быть опущен. В таком случае принимаемое по умолчанию значение параметра разновидности вещественного типа равно четырем. Разумеется, возможно использование и альтернативного способа объявления данных, например вместо REAL(4) можно использовать REAL*4. Также вещественные данные двойной точности могут быть объявлены оператором DOUBLE PRECISION. Вещественные данные представляются в ЭВМ в виде чисел с плавающей точкой.

Параметром разновидности может также быть именованная константа или возвращаемое функцией KIND значение. Так, объявление REAL(KIND(0.0)) эквивалентно объявлению REAL(4) или REAL(KIND = 4) (или REAL(8), если задана опция компилятора /4R8). Объявление REAL(KIND(0.0_8)) эквивалентно объявлению REAL(8) или REAL(KIND = 8). Задаваемая по умолчанию разновидность стандартного вещественного типа может быть изменена с 4 на 8 в результате использования опции компилятора /4R8 или директивы $REAL:8.

Диапазон изменения значений вещественных типов:

REAL(4) отрицательные числа: от -3.4028235E+38 до -1.1754944E-38;

число 0;
положительные числа: от +1.1754944E-38 до +3.4028235E+38;
дробная часть может содержать до шести десятичных знаков.

REAL то же, что и REAL(4)

REAL(8) отрицательные числа:
от -1.797693134862316D+308 до -2.225073858507201D-308;
число 0;
положительные числа:
от +2.225073858507201D–308 до +1.797693134862316D+308;
дробная часть может содержать до 15 десятичных знаков.

Пример 1:

```
integer(4), parameter :: m = 3, n = 5, low = 4
real(kind = 4) :: d(m, n) = 15.0, hot = 3.4
real(4), pointer :: da(:, :)
real(low) d2(n)
```

Пример 2. Атрибут может быть задан в виде оператора.

```
real(8) :: da                    ! Объявляем ссылочный массив da
pointer da                       ! Теперь, используя операторы, зададим атрибуты
dimension da(:, :)
parameter m = 10, n = 20
allocate(da(m, n))               ! Первый исполняемый оператор
```

3.2.3. Объявление данных комплексного типа

Комплексное число типа COMPLEX или COMPLEX(4) представляет собой упорядоченную пару вещественных чисел одинарной точности. Комплексное число типа COMPLEX(8) (DOUBLE COMPLEX) - упорядоченная пара вещественных чисел двойной точности. Например:

```
complex(4) :: c, z = (3.0, 4.0)
c = z / 2                        !       (1.50000, 2.00000)
```

Первый компонент пары представляет действительную, а второй - мнимую части числа. Оба компонента комплексного числа имеют одну и ту же разновидность типа.

Синтаксис оператора объявления объектов комплексного типа:

COMPLEX [([KIND =] *kind-value*)] [[, *attrs*] ::] *entity-list*

3.2.4. Объявление данных логического типа

Объекты логического типа объявляются оператором

LOGICAL [([KIND =] *kind-value*)] [[, *attrs*] ::] *entity-list*

Разновидность типа может принимать значения 1, 2 и 4 и совпадает с длиной логической величины в байтах. Задаваемую по умолчанию

разновидность стандартного логического типа данных LOGICAL можно изменить с 4 на 2, задав опцию компилятора /4I2 или директиву $INTEGER:2.

Пример:

```
logical, allocatable :: flag1(:), flag2(:)
logical(2), save :: doit, dont = .false._2
logical switch

! Эквивалентное объявление с использованием операторов вместо атрибутов
logical flag1, flag2
logical(2) :: doit, dont = .false._2
allocatable flag1(:), flag2(:)
save doit, dont
```

Логические величины (переменные, выражения) могут быть использованы в арифметических операторах и могут быть присвоены целым переменным.

Пример:

```
integer :: a = 2
logical :: fl = .true., g = .false.
write(*, *) a * fl, a * g              !         2    0
end
```

Правда, там, где требуются арифметические величины, например в спецификаторе UNIT оператора OPEN, применение логических величин недопустимо. Смешение логических и целых величин также недопустимо, если используется опция компилятора /4Ys.

3.3. Правила умолчания о типах данных

В Фортране допускается не объявлять объекты данных целого и вещественного типов. При этом тип данных объекта будет установлен в соответствии с существующими правилами умолчания: объекты данных, имена которых начинаются с букв *i, j, k, l, m* и *n* или с букв *I, J, K, L, M* и *N*, имеют по умолчанию стандартный целый тип (INTEGER); все остальные объекты имеют по умолчанию стандартный вещественный тип (REAL). Заметим, что на часть встроенных функций это правило не распространяется. Задаваемую по умолчанию разновидность типа данных можно изменить, задав для целого типа при компиляции опцию /4I2 или директиву $INTEGER:2 и для вещественного типа - опцию компилятора /4R8 или директиву $REAL:8 [1].

Пример:

```
integer :: x = 5                    ! Целочисленная переменная
y = 2 * x                           ! y - переменная типа REAL
```

3.4. Изменение правил умолчания

Изменение правил умолчания о типах объектов данных выполняется оператором IMPLICIT, который задает для объявленного пользователем имени принимаемый по умолчанию тип.

Синтаксис оператора:

IMPLICIT NONE

или

IMPLICIT *type(letters*) [, *type(letters*), ...]

type - один из встроенных или производных типов данных.

letters - список одинарных букв или диапазонов букв. Диапазон букв задается первой и последней буквой диапазона, разделенными знаком тире, например *c - f*. Буквы и диапазоны букв в списке разделяются запятыми, например:

```
implicit integer(4) (a, c - f), character(10) (n)
```

После такого задания все объекты данных, имена которых начинаются с букв *a* и *A* и с букв из диапазона *c - f* и *C - F*, будут по умолчанию иметь тип INTEGER(4), а объекты, имена которых начинаются с букв *n* и *N*, по умолчанию будут иметь тип CHARACTER(10).

Задание одной и той же буквы в операторе (непосредственно или через диапазон) недопустимо. Диапазон букв должен быть задан в алфавитном порядке. Знак доллара ($), который может использоваться в качестве первой буквы имени, следует в алфавите за буквой *Z*.

Оператор не меняет типа встроенных функций.

Явное задание типа имеет более высокий приоритет, чем тип, указываемый оператором IMPLICIT. Задание

IMPLICIT NONE

означает, что все используемые в программе имена должны быть введены явно (через операторы объявления типов данных). Невведенные имена приводят к возникновению ошибки на этапе компиляции. Никакие другие операторы IMPLICIT не могут указываться в программной единице, содержащей оператор IMPLICIT NONE. Ясно, что задание IMPLICIT NONE позволяет полностью контролировать типы всех объектов данных.

Пример:

```
implicit integer(a - b), character(len = 10) (n), type(feg)(c - d)
type feg
  integer e, f
  real g, h
end type
age = 10                          ! age - переменная типа INTEGER
name = 'Peter'                    ! name - переменная типа CHARACTER(10)
```

```
c%e = 1                    ! c%e - целочисленный компонент типа feg
$j = 5.0                   ! $j - Переменная типа REAL
```

3.5. Буквальные константы

В Фортране различают именованные и буквальные константы. Буквальные константы (далее - просто константы) используются в выражениях и операторах Фортрана. Возможно задание *арифметических, логических* и *символьных буквальных констант*.

3.5.1. Целые константы

Целые константы в десятичной системе счисления - целые числа (со знаком или без знака), например:

```
+2      2      -2
```

Константа может быть задана с указанием разновидности типа, значение которой указывается после значения константы и символа _ , например:

```
integer i*2, j*1
integer, parameter :: is = 1
i = -123_2                 ! Разновидность типа KIND = 2
j =  +123_is               ! Разновидность типа KIND = 1
write(*, *) i, j, kind(123_is)     !     -123      123      1
```

Для указания разновидности может быть использована ранее определенная именованная константа (в примере *is*) стандартного целого типа.

Целые константы по произвольному основанию задаются так:

[*знак*] [[*основание*] #] константа [_*kind*]

Знак это + или -. Для положительных констант знак может быть опущен. Основание может быть любым целым числом в диапазоне от 2 до 36. Если основание опущено, но указан символ #, то целое интерпретируется как число, заданное в шестнадцатеричной системе счисления. Если опущены и основание и символ #, то целое интерпретируется как имеющее основание 10. В системах счисления с основаниями от 11 до 36 числа с основанием 10, значения которых больше девяти, представляются буквами от *A* до *Z*. Регистр букв не является значащим. Константа может быть задана с указанием разновидности типа.

Пример. Представить десятичную константу 12 типа INTEGER(1) в системах счисления с основаниями 2, 10 и 16.

```
2#1100_1                   12_1 или 10#12_1#C_1 или 16#C_1
```

По умолчанию буквальные целые константы имеют стандартный целый тип.

Помимо названных возможностей, в операторе DATA беззнаковые целые константы могут быть представлены в двоичной, восьмеричной

или шестнадцатеричной форме с указанием системы счисления в виде символа, предшествующего значению константы (соответственно *B* или *b*, *O* или *o*, *Z* или *z* для двоичной, восьмеричной или шестнадцатеричной систем счисления). Сама же константа обрамляется апострофами (') или двойными кавычками. При этом константа должна использоваться для инициализации целой скалярной переменной.

```
integer i, j, k
data i /b'110010'/          ! Двоичное представление десятичного числа 50
data j /o'62'/              ! Восьмеричное представление числа 50
data k /z'32'/              ! Шестнадцатеричное представление числа 50
```

3.5.2. Вещественные константы

Вещественные константы используются для записи действительных чисел. Вещественные константы одинарной REAL(4) и двойной REAL(8) точности могут быть представлены в F-форме или в E-форме. Помимо этого, вещественные константы двойной точности могут быть представлены и в D-форме. Память, занимаемая вещественной константой одинарной точности, равна 4 байтам, а двойной точности - 8 байтам.

Вещественные константы в F-форме записываются в виде:

[+]|- [*целая часть*] . [*дробная часть*] [*_разновидность типа*]

Целая или дробная часть в F-форме может быть опущена, но не обе одновременно.

Пример:

```
+2.2      2.2_4      2.0_8      2.      -0.02_knd      -.02
```

Замечание. *knd* - ранее определенная константа стандартного целого типа (*knd* = 4 или *knd* = 8).

Константы в E-форме и D-форме имеют вид:

[+]|- [*мантисса*] E | e [+]|- *порядок* [*_разновидность типа*]

[+]|- [*мантисса*] D | d [+]|- *порядок*

Мантисса - число в F-форме или целое число.

Порядок - однозначное или двузначное целое положительное число.

Пример 1:

E- и D-формы числа $18.2*10^{11}$: +18.2E11 18.2e+11_8 18.2D11

E- и D-формы числа $-0.18*10^{-5}$: -.18E-05 -.18e-5 -.18d-5

Пример 2:

```
real(8) :: a*4 = +18.2E11, b = 18.2e+11_8, c /18.2D11/
! a - переменная одинарной, b и c - двойной точности
print *, a, b, c
```

Результат:

1.820000E+12 1.820000000000000E+012 1.820000000000000E+012

E- и D-формы также называются экспоненциальными формами числа и хороши для представления больших и малых чисел.

Вещественные константы одинарной точности могут представлять числа в диапазоне изменения значений типа REAL(4) (разд. 3.2.2). Дробная часть может содержать до шести десятичных знаков.

Вещественные константы двойной точности могут представлять числа в диапазоне изменения значений типа REAL(8) (разд. 3.2.2). Дробная часть может содержать до 15 десятичных знаков.

3.5.3. Комплексные константы

Комплексные константы используются для представления комплексных чисел и имеют вид:

[*знак*](*действительная часть*[_*разновидность*], &
мнимая часть[_*разновидность*])

Если *знак* задан, то он применяется как для действительной, так и для мнимой части. Задание общего знака при инициализации комплексной переменной в операторах DATA и объявлениях типов недопустимо.

Действительная (мнимая) часть - целая или вещественная константы.

Буквальные комплексные константы могут быть одинарной COMPLEX(4) и двойной COMPLEX(8) точности. В комплексных константах одинарной точности мнимая и действительная части занимают по 4 байта памяти; в комплексных константах двойной точности - по 8 байт. Поэтому комплексная константа одинарной точности занимает 8 байт памяти, а двойной - 16. При задании компонентов комплексной константы можно использовать одновременно для действительного и мнимого компонента константы E-, D- и F-формы. В случае одновременного использования одинарной и двойной точности при задании комплексной константы компилятор выполнит преобразование типов в соответствии с рангом типов арифметических операндов: комплексная константа будет иметь двойную точность, а компонент одинарной точности будет преобразован в форму с двойной точностью.

Пример 1. Число 36.8 - 263.3i в виде комплексной константы:

(36.8, -2.633E2)

или

(36.8, -263.3)

или

(36.8_4, -2.633E2_4)

Пример 2. D-форма числа $3.8*10^{-5}$ - $2.6*10^{-2}$i задает комплексную константу двойной точности:

(3.8D-5, -2.6D-2)

3.5.4. Логические константы

Логические константы используются для записи логических значений *истина* (.TRUE.) или *ложь* (.FALSE.). Отсутствие хотя бы одной обрамляющей точки в записи буквальной логической константы является ошибкой.

По умолчанию буквальные логические константы занимают в памяти ЭВМ 4 байта. Разновидность типа буквальной логической константы может быть задана явно, подобно тому, как это выполняется для буквальных целых констант. Например: .true._1 или .false._2.

Пример задания именованных логических констант:

```
logical(1), parameter :: fl = .true._1
logical(2) gl
parameter (gl = .false.)
```

3.5.5. Символьные константы

Символьные константы - последовательность одного или более символов 8-битового кода. Далее последовательность символов мы будем называть *строкой*. Символьные константы могут быть записаны с указателем длины и без него.

Символьные константы с *указателем длины*, называемые также *холлеритовскими константами*, имеют вид:

*n*H*последовательность символов*,

где *n* - целая константа без знака, задающая число символов в строке (ее длину); *H* (*h*) - буква, являющаяся разделителем между *n* и строкой. Число символов в *последовательности символов* должно быть равно *n*.

Пример:

```
18hthis is a constant
st = 16hthis is a string              ! Константа как элемент выражения
```

Символьная константа без указателя длины - это строка, заключенная в ограничители, апострофы или двойные кавычки. Ограничители вместе со строкой не сохраняются. Если строка должна содержать ограничитель, то она либо заключается в ограничители другого вида, либо ограничитель должен быть указан в строке дважды.

Пример:

```
'Это константа' или "Это константа"
"It's a constant" или 'It''s a constant'
```

Замечание. Символьные константы с указателем длины относятся к устаревшим свойствам Фортрана и не рекомендуются для применения.

Можно задать СИ-строковую константу. Для этого к стандартной строковой константе Фортрана необходимо прибавить латинские буквы *C* или *c*. Как известно, СИ-строки заканчиваются нулевым символом, имеющим в таблице ASCII код 0.

Пример СИ-константы: 'Это константа'c

В СИ-строке символы могут быть представлены в восьмеричном или шестнадцатеричном коде, которые указываются при задании констант после обратной наклонной черты. Например, '\62'c и '\x32'c задают символ '2' в восьмеричном и шестнадцатеричном кодах (ASCII-код символа '2' равен 50). Также в СИ существует специальная запись приведенных в табл. 3.2 часто используемых управляющих символов.

Таблица 3.2. Управляющие символы

Символ	*ASCII-код*	*Значение*
\0	0	Нулевой символ (*null*)
\a	7	Сигнал
\b	8	Возврат на шаг ("забой")
\t	9	Горизонтальная табуляция
\n	10	Новая строка
\v	11	Вертикальная табуляция
\f	12	Перевод страницы
\r	13	Возврат каретки
\"	34	Двойная кавычка
\'	39	Апостроф
\?	63	Знак вопроса
\\	92	Обратная наклонная черта
\ooo		Восьмеричная константа
\xhh		Шестнадцатеричная константа

Пример:

```
character :: bell = '\a'c              ! или '\007'c,  или '\x07'c
character(20) :: st = '1\a\a\t1-1\n\r2'c
write(*, *) bell                       ! Звуковой сигнал
write(*, *) st
```

Вывод строки *st* на экран произойдет так: в первой позиции начальной строки выведется символ '1'; затем прозвучат два звуковых сигнала; далее после выполнения табуляции в строке будут выведены символы '1-1'; после этого будет выполнен переход в первую позицию новой строки экрана и выведется символ '2'; далее последуют завершающие пробелы и *null*-символ.

В восьмеричном коде значение *o* находится в диапазоне от 0 до 7. В шестнадцатеричном коде *h* принимает значения от 0 до *F*.

При записи СИ-строк могут быть использованы двойные кавычки, например апостроф может быть задан так:

```
character quo /"\'"c/                ! или так: '\'"c
```

Также можно задать символьную константу нулевой длины.

```
character ch /''/                    ! '' - константа нулевой длины
print *, len(ch), len_trim(ch)       !     1      0
```

Поскольку символьные строки завершаются *null*-символом, то при их конкатенации (объединении) этот символ, если не принять специальных мер, окажется внутри результирующей строки, например:

```
character(5) :: st1 = 'ab'c, st2 = '12'c
character(10) res
res = st1 // st2                                  ! Вернет ab\0  12\0
print *, ichar(res(3:3)), ichar(res(8:8))          !     0      0
```

Длинная символьная буквальная константа, т. е. константа, которую не удается разместить на одной строке, задается с использованием символов переноса, например:

```
character(len = 255) :: stlong = 'I am a very, very, very long      &
&the longest in the world symbol constant (indeed very long -   &
&longer any constant you know)'
```

Замечание. В начале строки продолжения символ переноса может быть опущен.

3.6. Задание именованных констант

Защитить данные от изменений в процессе вычислений можно, задав их в виде именованных констант. *Именованная константа* - это именованный объект данных с атрибутом PARAMETER. Задание атрибута можно выполнить отдельным оператором:

PARAMETER [(] *name* = *const* [, *name* = *const* ...] [)]

или в операторе объявления типа:

typespec, PARAMETER [, *attrs*] :: *name* = *const* [, *name* = *const*] ...

typespec - любая спецификация типа данных.
name - имя константы. Не может быть именем подобъекта.

const - константное выражение. Выражение может включать имена констант, ранее введенных в той же программной единице. Допустимые операции константного выражения - арифметические и логические. Если тип константного выражения отличается от типа *name*, то любые операции преобразования типов выполняются автоматически.

attrs - иные возможные атрибуты константы.

Именованная константа может быть массивом или объектом производного типа. В первом случае для ее задания используется конструктор массива, во втором - конструктор производного типа.

При использовании оператора PARAMETER задание именованной логической, символьной и комплексной константы должно выполняться после описания ее типа. Типы целочисленных и вещественных констант могут быть установлены в соответствии с существующими умолчаниями о типах данных. Попытки изменить значение именованной константы при помощи оператора *присваивания* или оператора READ приведут к ошибке компиляции.

Именованная константа *name* не может быть компонентом производного типа данных, элементом массива и ассоциированным объектом данных, примененным, например, в операторах EQUIVALENCE или COMMON. Также именованная константа не может появляться в спецификации управляющего передачей данных формата. При использовании константы в качестве фактического параметра процедуры соответствующему формальному параметру следует задать вид связи INTENT(IN).

Пример 1. Задание именованных констант в операторе PARAMETER.

```
character(1) bell
parameter (bell = '\a'C)              ! СИ-строка
parameter (g = 9.81, pi = 3.14159)    ! Обрамляющие круглые скобки можно опустить
complex(4) z                          ! Сначала объявляется тип,
parameter (z = -(12.45, 6.784))       ! а затем задается значение
write(*, *) bell                      ! Звуковой сигнал
write(*, *) (bell, i = 1, 10)         ! Продолжительный сигнал
```

Пример 2. Использование PARAMETER в качестве атрибута.

```
program pa
 complex(4), parameter :: z = -(12.45, 6.784)
 integer(2), parameter :: a(5) = (/ 1, 3, 5, 7, 9 /)        ! a - массив-константа
 type made                                                  ! Задание типа made
  character(len = 8) bday
  character(len = 5) place
 end type made
 ! Задание константы pro типа made
 type(made), parameter :: pro = made('08/01/90', 'Mircu')
 write(*,'(1x, a10, 2x, a5)') pro
end program pa
```

3.7. Задание начальных значений переменных. Оператор DATA

В Фортране существует две возможности задания начальных значений переменных: в операторах объявления типа и оператором DATA. Начальные значения присваиваются переменным на этапе компиляции программы. Синтаксис оператора DATA:

DATA *список имен* /*список значений*/ &
[, *список имен* /*список значений*/] ...

Список имен - список переменных, их подобъектов и циклических списков. Элементы списка разделяются запятыми. Индексы элементов массивов и подстрок в *списке имен* должны быть целочисленными константными выражениями.

Список значений - список констант и/или повторяющихся констант, разделенных запятыми.

Повторяющаяся константа - элемент вида *n***val*, где *n* - целая положительная константа (буквальная или именованная); * - символ-повторитель. Такой элемент в списке значений означает, что *n* подряд расположенных переменных в списке имен получат в результате выполнения оператора DATA значение *val*.

Пример:

```
real(4) a(6, 7), d, r, eps, cmax
character st*6, sth*20, chr
integer(4) m, n
logical(1) flag, yesno
data  a /1, 2, 3, 4, 5, 6, 7, 35*0/,      &
      d, r /4, 6.7/,                      &
      eps /1.0e-8/, cmax /2.4e12/
data st /'Error!'/, chr /'Y'/, m, n /6, 7/
data sth /18hHollerith constant/
data flag, yesno /.true., .false./
```

При большом числе инициализируемых переменных следует для улучшения читаемости программы использовать строки продолжения или несколько операторов DATA.

Переменные производного типа инициализируются посредством применения в DATA конструктора производного типа (разд. 3.9.2.1) или путем инициализации их отдельных компонентов:

```
type pair
 real x, y
end type pair
type(pair) pt1, pt2                ! Используем для инициализации pt1
data pt1 / pair(1.0, 1.0) /        ! конструктор структуры
```

```
data pt2.x, pt2.y / 2.0, 2.0 /        ! Инициализации отдельных компонентов
print '(1x, 4f5.1)', pt1, pt2         !  1.0  1.0  2.0  2.0
```

Переменные, явно получившие атрибут AUTOMATIC, не могут появляться в операторе DATA.

При необходимости тип данных каждого числового или логического элемента в *списке значений* преобразовывается в тип, заданный для соответствующей переменной в *списке имен*. Например, для инициализации вещественной переменной можно использовать целую буквальную константу.

Число значений в каждом *списке значений* должно совпадать с числом элементов в соответствующем *списке имен*. Нельзя дважды в операторе DATA инициализировать одну и ту же переменную.

Инициализация элементов двумерного массива выполняется по столбцам, например:

```
real a(3, 2)
data a / 1, 2, 3, 4, 5, 6 /           ! Эта запись эквивалентна следующей
data a(1,1), a(2,1), a(3,1), a(3,2), a(2,2), a(3,2) /1, 2, 3, 4, 5, 6/
```

Если символьный элемент в *списке значений* короче, чем соответствующая переменная или элемент массива в *списке имен*, его размер увеличивается до длины переменной посредством добавления завершающих пробелов. Если же символьный элемент длиннее соответствующей переменной, то избыточные символы отсекаются.

Формальные параметры, переменные неименованных *common*-блоков и имена функций не могут появляться в операторе DATA. Переменные именованных *common*-блоков могут появляться в операторе DATA, если он используется в программной единице BLOCK DATA.

Оператор DATA может содержать в *списке имен* циклические списки:

(*dolist*, *dovar* = *start*, *stop* [, *inc*])

dolist - элемент массива, индексом которого является переменная *dovar*.

start, *stop*, *inc* - целочисленные константные выражения, определяющие диапазон и шаг изменения *dovar*. Если выражение *inc* отсутствует, то шаг устанавливается равным единице.

При использовании циклического списка можно выполнить инициализацию части массива. Возможна организация вложенных циклических списков.

Пример:

```
integer(4) a(20), b(5, 30), c(15, 15), row, col
integer, parameter :: rma = 10, cma = 5
data  (a(i), i = 4, 16, 2) / 4, 6, 8, 10, 12, 14, 16/          &
      ((b(i, j), j = 1, 12), i = 1, 2) /24 * -3/               &
      ((c(row, col), row = 1, rma), col = 1, cma) / 50 * 10/
```

При задании начальных значений переменных в операторах объявления типа начальное значение переменной следует сразу после объявления этой переменной. Возможно также, как и в случае оператора DATA, использование повторяющихся констант, например:

```
real   a(6, 7) /1, 2, 3, 4, 5, 6, 35*-1/,          &
       d /4/, r /6.7/,                             &
       eps /1.0e-8/, cmax /2.4e12/
character st*6/'Error!'/, chr/'Y'/, sth*20/18hHollerith constant/
integer m /6/, n /7/                  ! Ошибочна запись: integer m, n/6, 7/
! или, используя синтаксис Фортрана 90:
character(len = 6) :: st = 'Error!'
integer :: m = 6, n = 7               ! Наличие разделителя :: обязательно
```

Для инициализации переменных в операторах объявления типа могут использоваться конструкторы массивов и структур. Многомерный массив можно сконструировать из одномерного, применив функцию RESHAPE (разд. 4.12.4.3), например:

```
real(4) :: b(42) = (/ 1, 2, 3, 4, 5, 6, (-1, k = 7, 42) /)
real(4) :: c(6, 7) = reshape((/ 1, 2, 3, 4, 5, 6/ ), shape = (/ 6, 7 /), pad = (/ -1 /))
```

3.8. Символьные данные

3.8.1. Объявление символьных данных

Символьный тип данных в Фортране могут иметь переменные и константы, которые мы будем называть *строками*, а также массивы и функции. Элементом символьного массива является строка. Возвращаемый символьной функцией результат также является строкой.

Символьные объекты данных объявляются оператором CHARACTER:

CHARACTER [(*type-param*)] [[*attrs*] ::] *vname*

type-param - длина *vname* и значение параметра разновидности; может иметь одну из следующих форм:

- ([LEN =] *type-param-value*);
- (KIND = *expr*);
- (KIND = *expr*, LEN = *type-param-value*);
- ([LEN =] *type-param-value*, KIND = *expr*).

type-param-value - может быть либо звездочкой (*), либо целой константой без знака в диапазоне значений от 1 до 32767, либо целочисленным константным выражением, вычисляемым со значением в диапазоне от 1 до 32767. Также если оператор CHARACTER объявляет формальные параметры и размещен в теле оператора INTERFACE или в разделе объявлений процедуры, то для задания *type-param-value* можно использовать и неконстантное описательное выражение (см. разд. 5.6). При этом если соответствующий фактический параметр задан, то формальный

параметр не может иметь атрибут SAVE, появляться в операторе DATA или быть инициализирован в операторе CHARACTER. Например, формальный параметр *st*3 подпрограммы *sub*:

```
character(len = 15) :: st = 'example', st2*20 /'example_2'/
...
call sub(st, 15)
call sub(st2, 20)
...
end

subroutine sub(st3, n)
 integer(4) n
 character(len = n) st3          ! длина st3 при первом вызове
 print *, len(st3)               ! равна 15, а при втором - 20
 ...
end
```

Если значение выражения, определяющего длину символьного элемента, отрицательное, то объявляемые символьные элементы будут нулевой длины. Если значение *type-param* не задано, то по умолчанию длина символьного объекта данных принимается равной единице.

expr - целочисленное константное или описательное выражение, задающие разновидность символьного типа. Фортран поддерживает одно значение параметра разновидности (KIND = 1) для символьных объектов данных.

attrs - один или более атрибутов, разделенных запятыми. Если хотя бы один атрибут задан, наличие разделителя :: обязательно. Используются те же, что и с оператором INTEGER (разд. 3.2.1) атрибуты. Если задан атрибут PARAMETER, то необходимо и инициализирующее выражение, например:

```
character(len = 20), parameter :: st = 'Title'
```

vname - имя переменной, константы или функции (внешней, внутренней, операторной, встроенной).

По умолчанию строка, которой не задано начальное значение, состоит из *null*-символов. Поэтому функция LEN_TRIM вернет для этой строки ее полную длину. Полезно выполнять инициализацию строки, например, пробелами:

```
character(30) fn, path /' '/                ! path получает начальное значение
write(*, *) len(fn), len(path)              !      30        30
write(*, *) len_trim(fn), len_trim(path)    !      30         0
write(*, *) ichar(fn(1:1)), ichar(path(5:5))!       0        32
```

Если начальное значение строки содержит меньше символов, чем ее длина, то недостающие символы восполняются пробелами, которые

называются *завершающими*. Если начальное значение содержит больше символов, чем длина строки, то избыточные символы отсекаются.

Можно использовать символьные буквальные СИ-константы, завершаемые *null*-символами, например:

```
character(20) :: st = 'C string'c
write(*, *) st
```

Как и для других типов данных, можно использовать синтаксис оператора CHARACTER Фортрана 77, например:

```
character*15 st1 /'first'/, st2 /'second'/          ! Строки длиной в 15 символов
character*5 st3, st4*10, st5*15                     ! Строки длиной в 5, 10 и 15 символов
character*6 ast(10) /'Nick', 'Rose', 'Mike', 'Violet', 6*'???'/
character err*(*)
parameter (err = 'Error!')
```

Замечание. В FPS при инициализации с использованием коэффициента повторения необходимо следить за длиной инициализирующей буквальной символьной константы: ее длина должна совпадать с заданной в операторе CHARACTER длиной строки. Так, в операторе

```
character*6 ast(10) /'Nick', 'Rose', 'Mike', 'Violet', 6*'???'/
```

не будет выполнена инициализация последних пяти элементов массива. Для правильной инициализации следует использовать повторяющуюся константу 6*'???□□□', в которой символ □ означает пробел. В CVF таких проблем нет.

3.8.2. Применение звездочки для задания длины строки

Применение звездочки (*) для задания длины символьного объекта данных возможно в трех случаях:

1) при объявлении символьных именованных констант (объектов, имеющих атрибут PARAMETER). В этом случае длина строки равна числу символов константы, например:

```
character(*) st
! или character(len = *) st
! или character(len = *, kind = 1) st
! или character(*, kind = 1) st
! или character(kind = 1, len = *) st
parameter (st = 'exam')
```

или

```
character(len = *), parameter :: st = 'exam'! и так далее
```

Так же может быть объявлена символьная константа - массив:

```
character(len = *), parameter :: ast(3) = (/'jan', 'febr', 'march'/)
```

или

```
character(len = *), parameter :: ast(3) /'jan', 'febr', 'march'/
print *, len(ast)                ! 5 - длина каждого элемента массива
```

Длина элемента символьного массива вычисляется по максимальной длине инициализирующих массив символьных буквальных констант. В нашем примере такой константой является 'march';

2) звездочка (*) может быть применена для задания длины символьного элемента при объявлении формальных символьных параметров. В этом случае длина формального параметра равна длине фактического параметра. Например:

```
character(len = 15) :: st = 'example', st2*20 /'example_2'/
...
call sub(st)
call sub(st2)
...
end

subroutine sub(st3)
 character(len = *) st3          ! Длина st3 равна длине фактического параметра
 print *, len(st3)
...
end
```

3) при объявлении длины возвращаемого внешней нерекурсивной символьной функцией результата также может быть использована звездочка. В таком случае действительная длина результата определяется в операторе CHARACTER той программной единицы, в которой осуществляется вызов функции. Например:

```
integer, parameter :: n = 20
character(len = 4) :: ins = '+ - '
character(len = n) st, stfun          ! Длина возвращаемого функцией stfun
st = stfun(ins)                       ! результата равна n
print *, st                           ! # + - + - + - + - #
end

function stfun(pm)
 character(len = *) pm                ! Длина строки pm равна длине строки ins
! Длина возвращаемого символьной функцией результата определяется
! в той программной единице, где эта функция вызывается
 character(len = *) stfun
 character(len = len(stfun)) temp     ! Строка temp - пример
 temp = pm // pm //pm // pm           ! автоматического объекта данных
 stfun = '# ' // trim(temp) // ' #'   ! для таких объектов не могут быть
end                                   ! заданы атрибуты SAVE и STATIC
```

Символьные операторные или модульные функции, функции-массивы, функции-ссылки и рекурсивные функции не могут иметь спецификацию длины в виде звездочки (*).

3.8.3. Автоматические строки

Процедуры могут содержать не только строки, перенимающие размер от фактического параметра (в последнем примере это строка *pm*), но и локальные символьные переменные, размер которых определяется при вызове процедуры. В нашем примере это строка *temp*. Такие переменные относятся к *автоматическим объектам данных*, которые создаются в момент вызова процедуры и уничтожаются при выходе из нее. Автоматические объекты не должны объявляться с атрибутами SAVE или STATIC.

3.8.4. Выделение подстроки

Рассмотрим строку *st* из 10 символов со значением "Это строка". Подобно элементам массива, символы строки расположены в памяти компьютера один за другим (рис. 3.1).

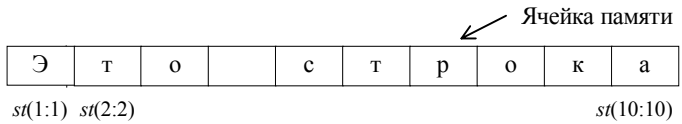


Рис. 3.1. Строка в памяти ЭВМ

Каждый символ строки имеет номер. Причем 1-й символ имеет номер 1, 2-й - номер 2 и т. д. Запись *st*(*i*:*i*) обеспечивает доступ к *i*-му символу строки.

Пример. Сколько раз буква 'т' содержится в строке *st*?

```
character(len = 20) :: st = 'Это строка'
integer(2) :: k, j
k = 0
do j = 1, len_trim(st)            ! Функция LEN_TRIM возвращает
  if(st(j:j) == 'т') k = k + 1    ! длину строки без завершающих
end do                            ! пробелов
print *, ' k =', k                ! Напечатает: k = 2
end
```

В Фортране можно выделить из строки любую ее *подстроку*:

st([*first*]:[*last*])

first - арифметическое выражение вещественного или целого типа, которое определяет первый символ в подстроке. По умолчанию значение *first* равно единице, и если *first* не задан, то подстрока начинается с первого символа строки.

last - арифметическое выражение вещественного или целого типа, которое определяет последний символ в подстроке. По умолчанию значение *last* равно длине строки, и если *last* не задан, то подстрока оканчивается последним символом строки *st*.

Замечания:

1. При необходимости дробная часть *first* (*last*) отбрасывается.

2. Значения *first* и *last* должны быть больше нуля; *last* не может превышать длину строки; *first* ≤ *last*.

3. Записи *st*(:) и *st* эквивалентны.

4. Операция выделения подстроки может быть также применена к элементу символьного массива и к символьному элементу записи. Например:

```
character st*20 /'It''s a string'/              ! st - строка; arrst - массив строк
character(len = 15) arrst(10) /'It''s a string  ', 9*'One more string'/
write(*, *) st(1 : 6)                          ! или st(:6)          It's a
write(*, *) arrst(2)(10 : 15)                  ! или arrst(2)(10:)    string
```

3.8.5. Символьные выражения. Операция конкатенации

Фортран содержит единственную символьную операцию - операцию *конкатенации* (//). Результатом операции является объединение строк - операндов символьного выражения. Длина результирующей строки равна сумме длин строк-операндов.

Операндами символьного выражения могут быть:

- символьные константы и переменные;
- символьные массивы и их сечения;
- элементы символьных массивов;
- вызовы символьных функций;
- символьные подстроки;
- символьные компоненты производных типов.

Пример:

```
character(len = 12) st1, st2                ! Строки длиной в 12 символов
character(len = 24) st
data st1, st2 /'first', 'second'/
print *, st1 // ' & ' // st2                ! first       & second
st = st1( : len_trim(st1)) // ' & ' // st2( : len_trim(st2))
print *, st                                 ! first & second
st = trim(st1) // ' & ' // trim(st2)
print *, st                                 ! first & second
```

Замечания:

1. Чтобы не потерять часть символов при объединении строк, надо следить, чтобы длина строки, которой присваивается результат конкатенации, была не меньше суммы длин объединяемых строк.

2. Функция LEN_TRIM вычисляет длину строки без завершающих пробелов.

3. Функция TRIM возвращает строку без завершающих пробелов.

4. Для выделения строки без завершающих пробелов применяется функция TRIM, например:

```
st = trim(sb)
```

В более ранних версиях то же самое выполнялось так:

```
st = sb(:len_trim(sb))
```

3.8.6. Присваивание символьных данных

Оператором присваивания в переменную символьного типа устанавливается результат символьного выражения:

символьная переменная = символьное выражение

Символьная переменная - строка, подстрока, символьный массив, символьные элементы записей. Длина *символьной переменной* может отличаться от длины строки - результата символьного выражения:

```
character st*9 /'Строка 1'/, stnew*14 /'Новая строка'/, st2 /'2'/
st = stnew                          ! 'Новая стро'
st = st2                            ! '2        '
st2 = stnew // st                   ! 'H'
```

Если *символьной переменной* является подстрока, то в результате присваивания изменяются принадлежащие подстроке символы:

```
character st*20 /'Строка 1'/, stnew*14 /'Новая строка'/, st2 /'2'/
st(8:) = stnew(1:5)                 ! 'Строка Новая'
st(14:14) = st2                     ! 'Строка Новая 2'
```

Присваивание символьных массивов и их сечений возможно, если они согласованы (разд. 4.6), например:

```
character(1) st(4) /'a', 'b', 'c', 'd'/, st2(4)
character(len=3) res(4)             ! Символьный массив из четырех элементов;
st2 = st                            ! длина каждого элемента массива равна трем
res = st // st2                     ! Все массивы согласованы
write(*, *) res                     ! aa bb cc dd
```

3.8.7. Символьные переменные как внутренние файлы

В Фортране символьная строка, подстрока и символьный массив являются *внутренними файлами*, т. е. файлами, существующими в оперативной памяти ЭВМ. В случае строки или подстроки этот файл имеет лишь одну запись, длина которой совпадает с длиной символьной переменной. В случае символьного массива число записей в файле равно числу его элементов. Для передачи данных при работе со строками как с внутренними файлами используются операторы В/В:

READ(*u*, *fms*) *список ввода*

WRITE(*u*, *fms*) *список вывода*

u - устройство В/В (см. разд. 10.2 и 10.3). При ссылке на внутренний файл номером устройства является имя символьной переменной, например строки, подстроки или символьного массива.

fms - спецификатор формата В/В, который в случае В/В под управлением неименованного списка задается в виде звездочки (*).

Используя оператор WRITE, в строку можно передать данные любых типов. И наоборот, оператором READ из строки можно считать, например, числовые данные (если в строке есть числовые поля данных). Часто при помощи оператора WRITE числовые данные преобразовываются в символьные, например число 123 в строку '123', а также формируются строки, состоящие из смеси числовых и символьных данных. Например, для обеспечения вывода сообщений, содержащих числовые данные, в графическом режиме посредством подпрограммы OUTGTEXT или при работе с диалоговыми окнами.

Пример. Преобразовать заданные числа (*a* и *b*) в символьное представление. Выполнить также обратное преобразование "строка - число".

```
integer(2) :: a = 123, a2
real(8) :: b = -4.56
character(10) sta, stb, ste, st*50
write(sta, '(A, I3)')  ' a = ', a
write(stb, '(F8.3)')   b              ! Номер устройства В/В совпадает с именем
write(ste, '(E10.4)') b               ! строки, в которую выполняется запись данных
print *, sta, stb, ste                ! a = 123     -4.560  -.4560E+01
write(st, '(A, I3, F7.2, E12.5)') ' a & b & b: ', a, b, b
print *, st                           ! a & b & b: 123  -4.56 -.45600E+01
read(st,'(12X,BZ,I3)') a2             ! Читаем из st значение a2
print *, a2                           ! 230
write(st, *) ' a : ', a               ! Вывод под управлением списка
print *, st                           ! a :        123
```

Пояснение. При чтении из строки (внутреннего файла) использованы преобразования: 12X - перемещение на 12 символов вправо, I3 - перевод последовательности символов '23 ' в число 230. Пробел интерпретируется как 0 благодаря дескриптору BZ.

3.8.8. Встроенные функции обработки символьных данных

Фортран содержит встроенные функции, позволяющие оперировать символьными данными. Встроенные функции разделяются на *элементные*, *справочные* и *преобразовывающие*. Аргументами элементных функций могут быть как скаляры, так и массивы. В последнем случае функция возвращает согласованный с массивами-параметрами массив. Значение элемента возвращаемого массива определяется в результате применения функции к соответствующему элементу массива-аргумента. Ряд функций,

например ICHAR или INDEX, возвращают значение стандартного целого типа INTEGER, имеющего по умолчанию параметр разновидности KIND = 4. Однако если применена опция компилятора /4I2 или директива $INTEGER:2, то тип INTEGER будет эквивалентен типу INTEGER(2) и, следовательно, функции стандартного целого типа, например INDEX, также будут возвращать значения типа INTEGER(2).

IACHAR(*c*) - элементная функция; возвращает значение стандартного целого типа, равное ASCII-коду символа *c*. Тип параметра *c* - CHARACTER(1).

Пояснение. Каждому символу поставлено в соответствие целое положительное число, называемое кодом символа. Американский стандарт обмена данными ASCII кодирует 128 символов, включающих управляющие символы, символы цифр, строчные и прописные буквы английского алфавита, знаки препинания и ряд других широко используемых символов. ASCII-коды символов расположены в диапазоне от 0 до 127. Первый ASCII-символ (символ с кодом 0) является пустым (*null*) символом - ''. Таким символом заканчивается любая СИ-строка. Коды со значениями, большими 127, употребляются для кодировки национальных алфавитов. Особенности кодировки русского алфавита рассмотрены в прил. 1.

ICHAR(*c*) - элементная функция; возвращает значение стандартного целого типа, равное коду символа *c* из поддерживаемой операционной системы таблицы символов. Тип параметра *c* - CHARACTER(1).

Пояснение. На практике операционная система может поддерживать отличную от ASCII кодировку символов. Получить доступ к системной таблице символов позволяет функция ICHAR. Правда, в Windows NT и Windows 95 последовательность символов с кодами от 0 до 127 совпадает с ASCII-последовательностью. Поэтому только для символов с кодами больше 127 IACHAR и ICHAR могут возвращать разные значения.

Пример. Вывести прописные буквы русского алфавита, коды которых в системной и ASCII-таблицах не совпадают.

```
character(1) caps(32) /'А', 'Б', 'В', 'Г', 'Д', 'Е', 'Ж', 'З',                  &
                       'И', 'Й', 'К', 'Л', 'М', 'Н', 'О', 'П', 'Р', 'С', 'Т', 'У',  &
                       'ф', 'Х', 'Ц', 'Ч', 'Ш', 'Щ', 'Ъ', 'Ы', 'Ь', 'Э', 'Ю', 'Я'/
integer :: n = 0                        ! Число различий
do i = 1, 32                            ! Сравнение кодов
 if(iachar(caps(i)) /= ichar(caps(i))) then
  print *, '  ', i, caps(i)
  n = n + 1
 end if
end do
if(n == 0) print *, 'Нет различий'
end
```

ACHAR(*i*) - элементная функция; возвращает символ типа CHARACTER(1), код которого в таблице ASCII-кодов символов равен *i* ($0 \le i \le 255$). Тип *i* - INTEGER.

CHAR(*i* [, *kind*]) - элементная функция; так же, как и функция ACHAR, возвращает символ типа CHARACTER(1), код которого в таблице ASCII-кодов символов равен *i* ($0 \le i \le 255$). Тип *i* - INTEGER. В отличие от ACHAR функция CHAR позволяет задать значение параметра разновидности символьного типа. Правда, в настоящее время символьный тип существует только с параметром разновидности KIND = 1. Значение параметра разновидности результата совпадает с *kind*, если параметр задан, и равно единице в противном случае.

Пример. Вывести на экран множество символов 8-битового кода, выводя на строчке по 15 символов. Напомним, что на 8 битах можно задать код для 256 символов с кодовыми номерами от 0 до 255.

```
do i = 1, 255, 15                    ! Вывод всех, кроме null, символов
  write(*, *) ('  ', char(j), j = i, i + 14)
end do
```

Замечание. Функции IACHAR, ICHAR, ACHAR, CHAR являются элементными, т. е. их аргументом может быть массив, например:

```
integer(4) iabc(5)
character(1) :: ABC(5) = (/ 'A', 'B', 'C', 'D', 'E' /)
iabc = ichar(ABC)
print *, iabc                  !     65     66     67     68     69
iabc = ichar((/'a', 'b', 'c', 'd', 'e'/))
print *, iabc                  !     97     98     99    100    101
end
```

LGE(*string_a*, *string_b*) - элементная функция; возвращает .TRUE., если строка *string_a* больше строки *string_b* или равна ей, иначе результат функции - .FALSE.

LGT(*string_a*, *string_b*) - элементная функция; возвращает .TRUE., если строка *string_a* больше строки *string_b*, иначе результат функции - .FALSE.

LLE(*string_a*, *string_b*) - элементная функция; возвращает .TRUE., если строка *string_a* меньше строки *string_b* или равна ей, иначе результат функции - .FALSE.

LLT(*string_a*, *string_b*) - элементная функция; возвращает .TRUE., если строка *string_a* меньше строки *string_b*, иначе результат функции - .FALSE.

Замечания:

1. Если сравниваемые параметры имеют разную длину, то при сравнении их длина выравнивается за счет дополнения более короткого параметра пробелами справа.

2. Сравнение выполняется посимвольно слева направо. Фактически сравниваются коды (ASCII) символов.

3. Параметрами функций LGE, LGT, LLE, LLT могут быть согласованные массивы. В этом случае результат может быть присвоен логическому массиву, согласованному с массивами-параметрами.

Пример. Вывести из списка фамилии, начинающиеся с буквы 'Н' или с последующих букв алфавита.

```
character(len = 20) group(30) /'Алферов', 'Салтыков', &
      'Новиков', 'Влазнев', 'Николаев', 25*'?'/
integer(2) i
do i = 1, 30
 if(lge(group(i), 'H')) then
  write(*, *) group(i)                          ! Салтыков
 end if                                         ! Новиков
end do                                          ! Николаев
end
```

Для полноты изложения отметим, что тот же результат будет получен и в случае применения обычной операции отношения ' ≥ ' - .GE. или >=:

```
if(group(i) >= 'H') write(*, *) group(i)
```

LEN(*string*) - справочная функция; возвращает длину строки *string*. Результат имеет стандартный целый тип. Задание значения строки *string* необязательно. Параметр *string* может быть символьным массивом. В этом случае функция возвращает длину элемента массива.

Пример:

```
character sta(20)*15, stb*20
write(*, *) len(sta), len(stb)                  !     15      20
```

LEN_TRIM(*string*) - элементная функция; возвращает длину строки *string* без завершающих пробелов. Результат имеет стандартный целый тип. Параметр *string* может быть символьным массивом.

Пример:

```
character(len = 20) :: stb = 'One more string'
character(len = 20) st, st2 /' '/
write(*, *) ' Len_trim_stb=', len_trim(stb)     !  Len_trim_stb = 15
write(*, *) len_trim( st ), len_trim( st2 )     !      20       0
write(*, *) len_trim('It''s a string    ')      !      13
```

ADJUSTL(*string*) - элементная функция; выполняет левое выравнивание символьной строки: удаляет все ведущие пробелы и вставляет их в конец строки, например:

```
print *, adjustl('     banana') // 'nbc'        ! banana     nbc
```

ADJUSTR(*string*) - элементная функция; выравнивает символьную строку по правой границе за счет удаления всех завершающих пробелов и их последующей вставки в начало строки, например:

```
print *, 'banana    ' // 'nbc'                    ! banana    nbc
print *, adjustr('banana    ') // 'nbc'          !    bananannbc
```

INDEX(*string*, *substring* [, *back*]) - элементная функция; возвращает номер позиции, с которой начинается первое вхождение строки *substring* в строке *string*. Результат имеет стандартный целый тип. Если параметр *back* отсутствует или задан со значением .FALSE., то поиск ведется слева направо. Если значение *back* есть .TRUE., то поиск ведется справа налево, т. е. начиная с конца строки. Если строка *substring* не содержится в строке *string*, то функция возвратит 0. Номер позиции в любом случае исчисляется от начала строки.

Пример:

```
character(120) sta /'Снег, снег, снег, снег, снег над тайгою... '/
print *, index(sta, 'снег')                     !    7
print *, index(sta, 'снег', .true.)             !   25
```

REPEAT(*string*, *ncopies*) - преобразовывающая функция; возвращает строку, содержащую *ncopies* повторений строки *string* (выполняет *ncopies* конкатенаций строки *string*), например:

```
character(10) st
st = repeat('Na', 5)                            !NaNaNaNaNa
```

SCAN(*string, set* [, *back*]) - элементная функция; возвращает номер позиции первого найденного в строке *string* символа строки *set*. Если логический параметр *back* отсутствует или задан со значением .FALSE., то выдается положение самого левого такого символа. Если *back* задан со значением .TRUE., то выдается положение самого правого такого символа. Функция возвращает 0, если в строке *string* нет ни одного символа строки *set*. Результат имеет стандартный целый тип.

Пример:

```
integer  array(2)
print *, scan('Fortran', 'tr')                  !    3
print *, scan('Fortran', 'tr', back = .true.)   !    5
print *, scan('FORTRAN', 'ora')                 !    0
array = scan((/'fortran', 'visualc'/), (/'a', 'a'/))
print *, array                                  !    6    5
```

Замечание. Когда функция SCAN используется с массивами, входящие в массив константы должны иметь одинаковую длину. Для выравнивания длины констант следует использовать пробелы, например:

```
array = scan ((/'fortran', 'masm   '/), (/'a', 'a'/))
print *, array                                  !    6    2
```

TRIM(*string*) - преобразовывающая функция; удаляет завершающие пробелы строки *string*, например:

```
print *, 'banana    ' // 'nbc'                    ! banana    nbc
print *, trim('banana    ') // 'nbc'             ! banananbc
```

VERIFY(*string*, *set* [, *back*]) - элементная функция; возвращает 0, если каждый символ строки *string* присутствует в строке *set*. В противном случае возвращает номер позиции символа строки *string*, которого нет в строке *set.* Результат имеет стандартный целый тип. Если логический параметр *back* отсутствует или задан со значением .FALSE., то выдается положение самого левого такого символа. Если *back* задан со значением .TRUE., то выдается положение самого правого такого символа, например:

```
write(*, *) verify ('banana', 'nbc')          !     2
write(*, *) verify ('banana', 'nbc', .true.)  !     6
write(*, *) verify ('banana', 'nba')          !     0
```

Замечания:

1. В Фортран 90 дополнительно включены функции: IACHAR, ACHAR, ADJUSTL, ADJUSTR, REPEAT и TRIM. Все остальные символьные функции были доступны в Фортране 77.

2. Вызов встроенных функций может быть выполнен с ключевыми словами (разд. 8.11.4), например:

```
print *, verify('banana', set = 'nbc', back = .true.)      !     6
```

3. Параметрами элементных функций INDEX, SCAN, VERYFY помимо скаляров могут быть массив и скаляр или согласованные массивы. В этом случае результат может быть записан в целочисленный массив, согласованный с массивом-параметром.

4. Параметром элементных функций ADJUSTL и ADJUSTR может также быть и массив. В этом случае результат может быть записан в символьный массив, согласованный с массивом-параметром.

Пример:

```
character(4), dimension(2) :: st = (/ 'abcd', 'dab'/), ch*1 / 'a', 'b' /
character(7), dimension(2) :: ast = (/ 'abcd   ', 'dab    '/), arst
integer(4) p(2)
p = index(st, 'a')
print '(2i4)', p                                  !  1   2
p = index(st, ch)
print '(2i4)', p                                  !  1   3
print '(i4)', len(st)                             !  4
arst = adjustr(ast)
print *, arst                                     !   abcd    dab
```

3.8.9. Выделение слов из строки текста

Рассмотрим часто решаемую при работе со строками задачу: выделение слов из заданной строки. Задачу рассмотрим в следующей формулировке: вывести каждое слово текстового файла на отдельной строке, считая, что слова разделяются одним или несколькими пробелами. Такая формулировка является упрощенной, поскольку в общем случае разделителями между словами могут быть сочетания знаков препинания и символов табуляции с пробелами.

Поясним содержание задачи примером. Пусть текстовой файл имеет имя 'c:\a.txt' и содержит текст:

```
Увы, на разные забавы
Я много жизни погубил!
```

Тогда результатом работы программы будет последовательность слов:

```
Увы,
на
разные
забавы
Я
...
```

Идея алгоритма выделения слова проста: найти позицию начала (*wb*) и конца (*we*) слова в строке и выделить слово как подстроку: *st*(*wb*:*we*). Данную процедуру повторять до тех пор, пока не будут проанализированы все строки текста.

```
character(len = 20) words(100)   ! Массив слов текста
character(len = 80) st           ! Строка текста
integer j, wb, we, nw, lst
integer, parameter :: unit = 1   ! Номер устройства, к которому
open(unit, file = 'c:\a.txt')    ! подсоединяется файл c:\a.txt
! Запишем все слова текста в массив words
nw = 0                           ! nw - число слов в тексте
do while(.not. eof(unit))        ! Цикл обработки строк
  read(unit, '(a)') st           ! Ввод строки текста
  write(*, *) st                 ! Контрольный вывод
  lst = len_trim(st)             ! Длина строки без завершающих пробелов
  wb = 0                         ! wb - начало текущего слова в строке
  ! Просмотрим поочередно все символы строки st.
  ! Если найден пробел, то возможны случаи:
  !  а) предыдущий символ отличен от пробела (wb > 0), следовательно,
  !    выполнен переход с конца слова на промежуток между словами
  !    и, зная начало (wb) и конец (we) текущего слова,
  !    мы можем добавить слово в массив words.
  !    Используем для этого подпрограмму addword.
  !    После добавления слова устанавливаем в wb нуль (wb = 0);
  !  б) предыдущим символом является пробел - выполняем переход на
```

```
  !     следующий символ.
  ! Если текущий символ отличен от пробела, то возможны варианты:
  !  а) мы находимся в начале строки или предыдущий символ является
  !    пробелом (wb = 0);
  !  б) предыдущий символ отличен от пробела (wb > 0) - выполняем
  !    дальнейшее перемещение по текущему слову.
  do j = 1, lst                        ! Просмотрим все символы строки
   if(st(j:j) == ' ') then
    if(wb > 0) call addword(words, st, wb, we, nw)
   else if(wb == 0) then               ! Обнаружено начало слова
    wb = j
    we = j
   else
    we = we + 1                        ! Перемещение по текущему слову
   end if
  end do
  ! После просмотра всей строки, если строка не была пустой,
  ! мы обязаны добавить последнее слово в массив words
  if(wb > 0) call addword(words, st, wb, we, nw)
end do
close(unit)
write(*, *) 'Число слов в тексте nw =', nw
do j = 1, nw
  write(*, *) words(j)
end do
end

subroutine addword(words, st, wb, we, nw)
 integer wb, we, nw
 character(len = *) words(*)        ! Перенимающий размер массив
 character(len = *) st              ! Строка, перенимающая длину
 nw = nw + 1
 words(nw) = st(wb : we)
 wb = 0
end
```

3.9. Производные типы данных

3.9.1. Объявление данных производного типа

Рассмотрим табл. 3.3, содержащую экзаменационные оценки.

Таблица 3.3. Экзаменационные оценки студентов

Ф. И. О.	*Экзамен 1*	*Экзамен 2*	*Экзамен 3*	*Экзамен 4*
Александров В. М.	4	5	3	4
Владимиров А. К.	3	5	4	2
...				

При работе с таблицей может возникнуть ряд задач: сохранить таблицу в файле; прочитать данные из файла; подсчитать среднюю оценку студентов; найти лучших (худших) студентов и т. д. При этом удобно при передаче данных в файл и считывании их из файла оперировать строками таблицы, имея доступ к отдельным элементам строки. Иными словами, при таком подходе строка таблицы должна быть самостоятельной переменной, состоящей из нескольких изменяемых компонентов. Такой переменной в Фортране является запись.

Запись - это переменная *производного* (*структурного*) типа данных. Записи вводятся оператором TYPE или, как в FPS версии 1, оператором RECORD.

Производный тип данных (*структура*) - это одно или несколько объявлений переменных (как правило, разного типа), сгруппированных под одним именем. *Структура* должна вводиться в разделе объявлений программы.

В Фортране 90 производный тип данных вводится оператором:

```
TYPE [, access-spec] [::] name
  [PRIVATE | SEQUENCE]
      component decl
      [component decl]
      ...
END TYPE [name]
```

access-spec - определяющий способ доступа к объявленному типу атрибут (PUBLIC или PRIVATE). Атрибуты PUBLIC и PRIVATE могут быть использованы только при объявлении типа в модуле (разд. 8.7). По умолчанию способ доступа PUBLIC (если только модуль не содержит оператор PRIVATE без указания в нем списка объектов). Задание атрибута PUBLIC означает, что тип и его не имеющие атрибута PRIVATE компоненты доступны во всех программных единицах, использующих модуль, в котором производный тип определен. Задание атрибута PRIVATE означает, что тип и/или его компоненты доступны только в модуле. Причем сам тип может иметь атрибут PUBLIC, а его компоненты - PRIVATE.

name - имя производного типа данных (структуры); оно не должно совпадать с именем другой переменной или функции, определенных в том же программном компоненте, также оно не может совпадать с именем встроенного типа данных, например COMPLEX. Имя структуры является *локальным* и поэтому структура как тип должна быть объявлена в каждой программной единице, в которой объявляются переменные введенного типа. Для уменьшения издержек на разработку программы рекомендуется объявлять структуры в отдельных модулях, ссылаясь на них в тексте программной единицы оператором USE.

component decl - любая комбинация одного или нескольких операторов объявления типов переменных, имеющая вид:

тип [[, *список атрибутов*] ::] *список компонентов*

В необязательном списке атрибутов могут присутствовать атрибуты POINTER и/или DIMENSION. Операторы объявления типов могут содержать скаляры и массивы встроенных и производных типов. При этом входящие в *component decl* операторы TYPE и/или RECORD должны ссылаться на ранее определенные производные типы. Фортран 95 позволяет операторам, входящим в *component decl*, содержать начальные значения переменных. Они по умолчанию будут являться начальными значениями соответствующих компонентов всех объектов этого типа. Инициализировать можно как все, так и отдельные компоненты. Например:

```
type entry                                  ! Объявление типа entry
  real(4) :: val = 3.0                      ! Инициализация компонента val
  integer(4) :: index                       ! Инициализация не выполняется
  type(entry), pointer :: next => null( )   ! Инициализация компонента next
end type entry
type(entry) :: erray(10)
print *, erray(5)%val                       !    3.000000
```

Если объявление производного типа содержит атрибут SEQUENCE, то его компоненты будут записаны в память в порядке их объявления в типе. Это позволяет использовать переменные производного типа в *common*-блоках, операторах EQUIVALENCE и в качестве параметров процедур.

Замечания:

1. Входящие в состав производного типа переменные называются его *компонентами*.

2. По умолчанию объявленный в модуле производный тип доступен в любой программной единице, использующей модуль.

3. При определении компонента производного типа могут быть использованы только два атрибута: POINTER и DIMENSION. При этом если компонент объявлен с атрибутом POINTER, то он может ссылаться на объект любого типа, включая и объект объявленного производного типа, например:

```
type entry                          ! Объявление типа entry
  real val
  integer index
  type(entry), pointer :: next      ! Ссылка на объект типа entry
end type entry
```

4. Следующий стандарт позволит задавать в производных типах размещаемые объекты данных.

После введения производного типа данных объекты (переменные или константы) нового типа объявляются оператором:

TYPE(*type-name*) [, *attrs*] :: *vname*

type-name - имя производного типа, введенного оператором TYPE ... END TYPE.

attrs - один или более разделенных запятыми атрибутов *vname*.

vname - одно или более разделенных запятыми имен переменных или констант, называемых *записями*. Присутствующее в *vname* имя может быть массивом.

Оператор TYPE, как и другие операторы объявления данных, предшествует исполняемым операторам. Оператор должен быть расположен после введения типа *type-name*.

Запись является составной переменной. Для доступа к компоненту записи используется *селектор компонента* - символ процента (%) или точка (последнее невозможно при задании директивы $STRICT):

val = vname%cname или val = vname.cname

где *cname* - имя компонента записи. Если компонент *cname* является записью, то для доступа к компоненту *cname* потребуется дважды применить селектор компонента:

val2 = vname%cname%cname2

где *cname2* - имя компонента записи *cname*. И так далее.

Пример:

```
integer, parameter :: n = 20                ! Можем использовать n внутри
character(n) bname                          ! объявления производного типа
type catalog                                ! Описание каталога
 character(n) name, phone                   ! Название, телефон
 integer cat_id                             ! Код каталога
end type catalog
type(catalog) boa                           ! Объявление записи
boa = catalog('JCP', '234-57-22', 44)       ! Конструктор структуры
bname = boa % name                          ! Доступ к компоненту записи
print *, bname, '  ', boa%phone             ! JCP  234-567-22
```

Замечание. Из примера видно, что заданную до описания производного типа именованную константу можно использовать в объявлении этого типа, например для задания длины символьной строки.

3.9.2. Инициализация и присваивание записей

3.9.2.1. Конструктор производного типа

Переменную производного типа можно определить (присвоить значения всем ее компонентам), применив *конструктор производного типа*, называемый также *конструктором структуры*:

имя-типа (*список выражений*)

где *список выражений* задает значение компонентов переменной.

Конструктор структуры может быть использован для инициализации записей в операторах объявления записей, в операторе DATA, в операторе присваивания, в выражениях (если выполнена перегрузка операций) и в качестве фактического параметра процедуры.

Аналогичный *конструктор* используется и для генерации констант производного типа:

имя-типа (*список константных выражений*)

Пример. Сформируем структуру *order*, содержащую информацию о заказе покупателя. Каждый заказ может содержать до 10 наименований вещей.

```
type item_d                          ! Описание заказанной вещи
 character(20) descr, color, size    ! Название, цвет, размер
 integer(2) qty                      ! Количество
 real(4) price                       ! Цена
end type
type order                           ! Описание заказа
 integer(4) ordnum, cus_id           ! Номер заказа, код покупателя
 type(item_d) item(10)               ! Переменная типа item_d
end type
! Задание записи - константы
type(order), parameter :: e_ord = order(1, 1, item_d('d', 'c', 's', 1, 1.0))
type(order) cur_ord                  ! Переменная типа order
! Используя конструктор структуры, занесем в заказ cur_ord
! 10 одинаковых вещей
! Одним из выражений конструктора order является конструктор item_d
cur_ord = order(1200, 300, item_d('shorts', 'white', 'S', 1, 35.25))
print *, cur_ord%item(1)             ! Вывод данных о первом предмете
! Для вывода цвета вещи потребуется дважды применить селектор компонента
print *, cur_ord%item(2)%color  ! Вывод цвета второго предмета
```

Замечания:

1. Поскольку переменная типа *item_d* входит в состав типа *order*, то тип *item_d* должен быть введен до описания типа *order*.

2. Определить переменную *cur_ord* можно, предварительно определив массив *item*(10). Выполним это в операторе объявления записи:

```
type(item_d) :: item(10) = item_d('shorts', 'white', 'S', 1, 35.25)
type(order) cur_ord
cur_ord = order(1200, 300, item)
print *, item(1)                     ! Вывод данных о первом предмете
print *, cur_ord%ordnum              ! Вывод номера заказа
```

3.9.2.2. Присваивание значений компонентам записи

Продолжим работу с переменной *cur_ord* только что введенного производного типа *order*:

```
! Присвоим значение отдельному компоненту записи
cur_ord%cus_id = 1300                    ! Изменим код покупателя
! Присвоим значение компоненту записи - элементу массива:
cur_ord%item(2)%color = 'blue'           ! Изменим цвет второй вещи заказа
! Присвоим значение всему массиву - компоненту записи:
cur_ord%item%color = 'none'
```

Если компонентом записи является массив, то для его определения можно использовать конструктор массива (разд. 4.6), например:

```
type vector
 integer n
 integer vec(10)
end type                                 ! j-му элементу массива vec присвоим значение j * 2
type(vector) :: vt = vector(5, (/ (j * 2, j = 1, 10) /))
print *, vt.n, vt.vec(2)                 !        5        4
```

3.9.2.3. Задаваемые присваивания записей

Можно изменить значение переменной производного типа, присвоив ей значение другой переменной, константы, конструктора или выражения того же типа. Однако область действия встроенного присваивания можно расширить, связав с оператором присваивания (=) посредством блока INTERFACE ASSIGNMENT модульную или внешнюю подпрограмму, которая будет вызываться каждый раз, когда в программе встречается заданное присваивание (разд. 8.12.2).

3.9.3. Выражения производного типа

Если не принять специальных мер, нельзя применять встроенные операции в выражениях с записями. Нельзя, например, сложить две записи, применяя встроенную операцию сложения. Мерой, позволяющей распространить встроенную операцию на производный тип, является перегрузка операций (разд. 8.12.2). Для задания (перегрузки) операции создается функция, которая при помощи интерфейсного блока связывается с задаваемой операцией. Эта функция вызывается каждый раз, когда встречается заданная операция, и возвращает для последующего использования в выражении результат этой операции.

Пример. Зададим операцию умножения числа на запись.

```
module deta                              ! Определим производный тип pair
 type pair                               ! в модуле deta
  real x, y
 end type pair
end module
```

```
program paw
 use deta                          ! Получаем доступ к типу pair
 interface operator(*)             ! К задающей операцию внешней функции
  function mu(a, b)                ! необходимо явно описать интерфейс
   use deta
   type(pair) mu
   type(pair), intent(in) :: b     ! Вид связи параметров задающей
   real, intent(in) :: a           ! операцию функции должен быть IN
  end function
 end interface
 type(pair) :: pt1 = pair(2.0, 2.0), pt2
 pt2 = 2.0 * 2.5 * pt1             ! Первая операция умножения встроенная,
                                   ! вторая - перегруженная
 print *, pt2                      !       10.000000       10.000000
end program paw
function mu(a, b)                  ! Функция будет вызываться каждый раз,
 use deta                          ! когда первым операндом операции *
 type(pair) mu                     ! будет выражение типа REAL,
 type(pair), intent(in) :: b       ! а вторым - выражение типа pair
 real, intent(in) :: a
 mu.x = a * b.x
 mu.y = a * b.y
end function
```

3.9.4. Запись как параметр процедуры

Если запись используется в качестве параметра процедуры и ее тип повторно определяется в процедуре оператором TYPE ... END TYPE, то при его определении и в вызывающей программной единице, и в процедуре необходимо задать атрибут SEQUENCE. Это обеспечит одинаковое расположение компонентов записи в памяти. (Порядок размещения компонентов в памяти определяется на этапе компиляции.) Если в определении производного типа встречаются иные производные типы, то они тоже должны иметь атрибут SEQUENCE.

Если производный тип определяется в модуле, атрибут SEQUENCE избыточен: модули компилируются отдельно и, следовательно, в каждой программной единице, получающей определение производного типа посредством *use*-ассоциирования, компоненты такой записи будут размещены в памяти одинаково.

Пример:

```
program gopo                       ! Главная программа
 type point                        ! В главной программе и функции pval
  sequence                         ! определен один и тот же тип point
  real x, y
 end type
 type(point) pt
```

```
 call pval( pt )
 print *, pt                    !       1.000000        -2.000000
end program gopo
subroutine pval(pt)
 type point
  sequence
  real x, y
 end type
 type(point) pt
 pt.x = 1.0
 pt.y = -2.0
end subroutine
```

Два определения типа в разных программных единицах определяют один и тот же тип, если оба имеют одно и то же имя, обладают атрибутом SEQUENCE, их компоненты не являются приватными и согласуются в отношении порядка их следования, имен и атрибутов. Однако более рациональным представляется однократное описание производного типа в модуле с последующей ссылкой на модуль в программных единицах, использующих этот тип.

Атрибут SEQUENCE также должен быть использован при размещении записи в *common*-блоке, например:

```
program gopo
 type point
  sequence
  real x, y
 end type
 type(point) pt
 real s, t
 common /a/ s, pt, t
 call pval( )
 print '(4f5.1)', s, pt, t          !  2.0  1.0 -2.0 -1.0
end program gopo
subroutine pval( )
 common /a/ s, x, y, t
 s = 2.0; t = -1.0
 x = 1.0                        ! x и y определяют компоненты
 y = -2.0                       ! записи pt главной программы
end subroutine
```

3.9.5. Запись как результат функции

Результат функции может иметь производный тип, например внешняя функция *mu* (разд. 3.9.3) возвращает значение типа *pair*.

При задании *внешней* функции производного типа следует описать этот тип как внутри функции, так и в каждой вызывающей функцию программной единице. Лучше всего для этих целей определить тип в

модуле и затем использовать *use*-ассоциирование. При явном определении типа и в функции, и в вызывающих ее программных единицах потребуется использование атрибута SEQUENCE. Сама же функция должна быть объявлена в каждой вызывающей ее программной единице.

Если же имеющая производный тип функция является *внутренней*, то этот тип может быть описан только в программной единице, из которой эта функция вызывается. При этом тип функции определяется только в самой функции, например:

```
module deta
 type pair
  real x, y
 end type pair
end module deta

program paw2
 use deta                          ! Получаем доступ к типу pair
 type(pair) :: pt1 = pair(2.0, 2.0)
 type(pair) :: pt2
 pt2 = mu(2.5, pt1)                ! Вызов внутренней функции mu
 print *, pt2                      !        5.000000        5.000000

 contains

  function mu(a, b)                ! Внутренняя функция типа pair
   type(pair) mu                   ! Объявление типа функции
   type(pair), intent(in) :: b
   real, intent(in) :: a
   mu.x = a * b.x
   mu.y = a * b.y
  end function mu
end program paw2
```

3.9.6. Пример работы с данными производного типа

Рассмотрим пример, иллюстрирующий механизм передачи записей из программы в файл и обратно из файла в программу. Пусть файл c:\exam.dat содержит данные о результатах экзаменационной сессии студенческой группы. (Для генерации файла c:\exam.dat в программе использован датчик случайных чисел.) Каждой записью файла является строка табл. 3.3. Вывести на экран из созданного файла все его записи и среднюю оценку студентов.

```
module tex
 type exam                         ! Структура exam
  character(30) name               ! Студент
  integer(4) m1, m2, m3, m4        ! Экзаменационные оценки
 end type
 type(exam) stud                   ! stud - переменная типа exam
 integer :: unit = 2               ! Номер устройства подсоединения
```

```
end module tex                    ! файла c:\exam.dat
program aval
use tex                           ! Включаем описание структуры
integer(4) :: ns = 20             ! Число студентов в группе
real(4) :: am = 0.0               ! Средняя оценка студентов
! Открываем двоичный файл
open(unit, file = 'c:\exam.dat', form = 'binary')
call testfile(ns)                 ! Наполняем файл exam.dat
rewind unit                       ! Переход на начало файла
do while(.not. eof(unit))         ! Обработка данных файла
 read(unit) stud
 am = am + stud%m1 + stud%m2 + stud%m3 + stud%m4
 write(*, '(1x, a20, 4i4)') stud  ! Контрольный вывод
end do
close(unit)
am = am / float(ns * 4)
write(*, *) ' Средняя оценка группы: ', am
end
subroutine testfile(ns)
 use tex                          ! Включаем описание структуры
 integer ns, i
 integer sv
 write(*, '(1x, a $)') 'Старт random (INTEGER*4): '
 read(*, *) sv
 call seed(sv)
 do i = 1, ns
  ! Имя студента имеет вид: Name номер, например, Name 01
  write(stud%name, '(a, i3.2)') 'Name ', i
   stud%m1 = rmark( )             ! Генерируем экзаменационные оценки
   stud%m2 = rmark( )             ! Оценка - случайное число от 2 до 5
   stud%m3 = rmark( )
   stud%m4 = rmark( )
   ! Последние 4 оператора можно заменить одним:
   ! stud = exam(stud%name, rmark( ), rmark( ), rmark( ), rmark( ))
  write(unit) stud                ! Добавляем запись в файл
 end do
 contains
 integer function rmark( )        ! Генератор экзаменационных оценок
  real(4) rnd
  call random(rnd)                ! rnd - случайное число типа REAL(4) (0.0 ≤ rnd < 1.0)
  rmark = nint(8.5 * rnd)         ! Округление
  rmark = max(rmark, 2)           ! Оценка не может быть менее двух
  rmark = min(rmark, 5)           ! Оценка не может быть более пяти
 end function
end
```

Пояснения:

1. В Фортране структурная переменная может быть записана "целиком" как в неформатный (двоичный), так и в текстовой файл (в более ранних версиях Фортрана в текстовом файле запись можно было сохранить лишь покомпонентно). При передаче записи *stud* в текстовой файл можно использовать, например, такой форматный вывод (пусть файл подсоединен к устройству 3):

```
write(3, '(1x, a30, 4i3)') stud
```

В задаче для хранения данных использован последовательный двоичный файл exam.dat. Передача в файл осуществляется в подпрограмме *testfile*. Каждая запись файла имеет вид:

Name *номер* Оценка 1Оценка 2Оценка 3Оценка 4

Начальное значение *номера* - 01. Строка является внутренним файлом; поэтому проще всего получить строку вида Name *номер*, записав в символьную переменную *stud%name* данные 'Name ', *номер* по формату '(*a*, *i*3.2)', где *номер* меняется от 1 до *ns*. Каждая оценка формируется случайным образом в диапазоне от 2 до 5 функцией *rmark*, которая, в свою очередь, использует генератор случайных чисел (от 0.0 до 1.0) - встроенную подпрограмму RANDOM. Формируемая последовательность оценок зависит от начальной установки RANDOM, которая определяется значением параметра подпрограммы SEED.

2. Символ $ в спецификации формата оператора WRITE(*, '(1*x*, *a* $)') обеспечивает вывод без продвижения, что позволяет ввести значение *sv* на той же строке, где выведено сообщение '*Cmapm random* (INTEGER*4): '.

3. Выход из цикла происходит при достижении конца файла (функция EOF вырабатывает .TRUE.).

В результате работы программы на экран будут выведены строки (последовательность оценок зависит от значения параметра *sv*):

```
Name 01   2   4   5   5
...
Name 20   3   5   5   4
```

4. Чтобы избежать повторного описания структуры в главной программе и подпрограмме *testfile*, ее описание выполнено в отдельном модуле *tex*, который затем включается в текст программных единиц оператором USE.

3.9.7. Структуры и записи

3.9.7.1. Объявление и присваивание значений

Фортран CVF и FPS наследует от предшествующих версий еще одну возможность объявления производного типа данных - оператор STRUCTURE, которым, правда, не следует пользоваться при написании нового кода. Синтаксис оператора:

STRUCTURE /*имя структуры*/
объявление компонентов структуры
END STRUCTURE

Имя структуры - имя нового типа данных, оно не должно совпадать с именем другой переменной или функции, определенных в том же программном компоненте; также оно не может совпадать с именем встроенного типа данных, например COMPLEX.

Объявление компонентов структуры - любая комбинация одного или нескольких операторов объявления типов переменных или конструкций UNION. Операторы объявления типов могут содержать простые переменные, массивы, строки и операторы RECORD, которые ссылаются на ранее определенные структуры. Элементы структуры объявляются без атрибутов.

Имя структуры является *локальным*, и поэтому структура как тип должна быть объявлена (явно или в результате *use*-ассоциирования) в каждой программной единице, в которой необходимо работать с переменными введенного типа.

Структуры, содержащие оператор RECORD, называются *вложенными*. Вложенные структуры могут содержать компоненты с одинаковыми именами.

Длина структуры не может превышать 64 Кбайт. Способ упаковки структуры в памяти контролируется директивой $PACK и параметром /Zp в командной строке компилятора. Структуры являются одинаковыми, если их компоненты имеют одинаковый тип и размещены в структуре в одной и той же последовательности. Кроме этого, они должны иметь одинаковую упаковку в памяти ЭВМ.

Переменные структурного типа объявляются оператором

RECORD /*имя структуры*/ [, *attrs*] [::] *vname*

attrs и *vname* имеют тот же смысл, что и в операторе объявления производного типа TYPE.

Имя структуры должно быть введено до применения оператора RECORD. Оператор RECORD должен предшествовать исполняемым операторам программного компонента.

Пример:

```
structure /item_d/                ! Описание заказанной вещи
  character*20 descr, color, size ! Название, цвет, размер
  integer*2 qty                   ! Количество
  real*4 price                    ! Цена
end structure
structure /order/                 ! Описание заказа
 integer*4 ordnum, cus_id         ! Номер заказа, код покупателя
 record /item_d/  item(10)        ! Переменная типа item_d
```

```
end structure
record /order/ cur_ord              ! Переменная типа order
cur_ord = order(1200, 300, item_d('shorts', 'white', 'S', 1, 35.25))
print *, cur_ord.item(1)            ! Вывод данных о первом предмете
print *, cur_ord.item(2).color      ! Вывод цвета второй вещи
```

Запись состоит из компонентов, определенных оператором STRUCTURE. Доступ к компоненту записи осуществляется посредством указания после имени структурной переменной (записи) точки или знака % и имени компонента, например:

```
c_id = cur_ord.cus_id               ! Код покупателя
i_color = cur_ord.item.color        ! Цвет изделия
```

или

```
c_id = cur_ord%cus_id
i_color = cur_ord%item%color
```

Компоненты записи не имеют отличий от других переменных, имеющих тот же тип, кроме одного: целочисленный элемент записи не может быть использован в качестве переменной (*dovar*) DO-цикла.

При работе с текстовыми файлами можно выполнять форматный или управляемый списком В/В как компонентов записи, так и всей записи.

3.9.7.2. Создание объединений

В ряде задач необходимо записывать в файл (или считывать из файла) последовательно одинаковые по размеру, но разные по составу записи. В Фортране имеется возможность выполнять эти действия, работая с одной и той же структурой. Для этих целей следует задать структуру, в которой разрешено разным группам данных занимать одну и ту же область памяти. Группа данных оформляется оператором MAP. А объединение групп и отображение на одну и ту же область памяти задается оператором UNION. Операторы имеют синтаксис:

```
MAP
 объявление элементов структуры
END MAP

UNION
 map-блок
 map-блок
 [map-блок ...]
END UNION
```

Внутри объединения должно быть не менее двух *map*-блоков. Блок *union* может находиться только внутри оператора STRUCTURE. Блок *map* может находиться только внутри оператора UNION. Объединяемые *map*-блоки

должны иметь одинаковый размер. Аналогом содержащей объединения структуры, например в Паскале, является запись с вариантными полями.

Пример:

```
structure sam
 union
  map
   character*20 string
  end map
  map
   integer*2 number(10)
  end map
 end union
end structure
```

3.9.8. Итоговые замечания

Производный тип данных (структуру), если не нужно создавать объединения, следует вводить оператором TYPE ... END TYPE, используя затем для объявления записи (переменной производного типа) оператор TYPE. Компонентом записи, наряду с простыми переменными, могут быть и массивы и другие записи.

Записи, наряду со строками и массивами, относятся к составным переменным. Имеющие ранг 0 записи являются скалярами. Можно так же, как и в случае данных встроенных типов, объявить записи-массивы.

Запись считается неопределенной, если не определен хотя бы один ее компонент. Инициализация записи, создание записи-константы, присваивание записи значения выполняются посредством конструктора структуры, который позволяет определить или изменить значение всех компонентов записи. Кроме этого, используя селектор компонента, можно менять значение одного элемента записи. Отдельные компоненты записи могут быть использованы в выражениях так же, как и другие объекты встроенных типов.

Записи можно присвоить результат выражения того же типа.

Запись может быть "целиком" записана как в двоичный, так и в текстовой (форматный) файл (или считана из таких файлов).

Запись или массив записей могут быть использованы в качестве параметра процедуры, но при этом тип, к которому принадлежит запись, должен быть объявлен как в вызывающей программной единице, так и в вызываемой процедуре. При объявлении типа записи-параметра используется атрибут SEQUENCE. Такой же атрибут используется и при размещении записи в *common*-блоке.

Можно создать функцию, результатом которой является запись.

Используя оператор INTERFACE OPERATOR, можно перегрузить встроенную или создать новую операцию (унарную или бинарную),

операндами которой являются записи. Наличие такой возможности улучшает структуру и сокращает код программы.

Используя оператор INTERFACE ASSIGNMENT, можно задать присваивание, в котором компонентам переменной производного типа по определенным правилам присваивается результат выражения другого типа.

3.10. Целочисленные указатели

Для доступа к занимаемой переменными памяти, а также к свободной памяти ЭВМ в CVF и FPS используются целочисленные указатели.

Целочисленный указатель - это переменная, содержащая адрес некоторой ячейки памяти и связанная с некоторой переменной, называемой *адресной переменной*. Целочисленный указатель имеет 3 компонента: собственно указатель, связанную с ним адресную переменную и объект-адресат, адрес которого содержит указатель. Объектом-адресатом может также быть и ячейка выделенной функцией MALLOC памяти. Задание указателя выполняется в два этапа. Первоначально указатель связывается с адресной переменной. Это выполняется посредством оператора POINTER, имеющего синтаксис

POINTER(*pointer*, *varname*) [(*pointer*, *varname*)]...

Пример:

```
real var
pointer(p, var)                    ! p - указатель; var - адресная переменная
```

Целочисленный указатель всегда имеет тип INTEGER(4) и не должен объявляться явно. Связываемая с ним адресная переменная (скалярная или массив) может быть любого типа.

Оператор POINTER должен располагаться в разделе объявлений программной единицы.

На втором этапе в указатель устанавливается адрес некоторого объекта или ячейки памяти. В первом случае это выполняется функцией LOC, во втором - MALLOC. Например:

```
real var(5), a(5)
pointer(p, var)
p = loc(a)
```

Адресная переменная через указатель связана с областью памяти, адрес которой установлен в указатель, и обладает двумя свойствами:

- устанавливаемые в адресную переменную значения размещаются по хранящемуся в указателе адресу и, таким образом, передаются объекту-адресату;
- данные, хранящиеся по содержащемуся в указателе адресу, передаются в связанную с ним адресную переменную.

Пример первого свойства адресной переменной:

```
real(4) a(5) /5*0.0/, var(5), wa
pointer(p, var) (p2, wa)        ! Объявим два указателя p и pa и свяжем первый
integer k                       ! с адресной переменной var, а второй - с wa
p = loc(a)                      ! Установим указатель p на начало массива a
var(2) = 0.5                    ! Устанавливает также и в a(2) значение 0.5
print '(10f5.1)', a             !  .0   .5   .0   .0   .0
var = -1.3                      ! Все элементы массива a равны -1.3
print '(10f5.1)', a             !-1.3 -1.3 -1.3 -1.3 -1.3
p2 = loc(a)                     ! Установим указатель p2 на начало массива a
do k = 1, 5
 wa = float(k)                  ! В a(k) устанавливается значение FLOAT(k)
 p2 = p2 + 4                    ! Тип a - REAL(4), поэтому для перехода к следующему
end do                          ! элементу массива a выполним p2 = p2 + 4
print '(10f5.1)', a             ! 1.0  2.0  3.0  4.0  5.0
end
```

Пример использования указателя с символьным типом данных. Заменить в заданной строке каждый нечетный символ на букву *b*.

```
character(20) :: st = '1#2#3#4#5#6#7#8#9#', ch*1
pointer(p, ch)
integer p0
p0 = loc(st)                    ! Установим в p0 адрес начала строки st
do p = p0, p0 + len_trim(st), 2
 ch = 'b'                       ! Каждый символ строки занимает 1 байт, поэтому
end do                          ! используем шаг 2 для перемещения по нечетным символам
print *, st                     ! b#b#b#b#b#b#b#b#b#
end
```

Пример второго свойства адресной переменной:

```
real(4) a(5) /1.0, 2.0, 3.0, 4.0, 5.0/, wa
pointer(p, wa)
integer p0          ! В адресную переменную wa передается содержимое
p0 = loc(a)         ! области памяти, которую адресует указатель p
print '(10f5.1)', (wa, p = p0, p0 + 19, 4)        ! 1.0  2.0  3.0  4.0  5.0
end                 ! Указатель p использован в качестве параметра цикла
```

Указатель может быть использован в качестве фактического параметра процедуры, в которой может быть изменено его значение. В общем случае целочисленный указатель может появляться везде, где могут использоваться целочисленные переменные. Следует только помнить, что указатель содержит адрес объекта, и следить за тем, чтобы после всех изменений значения указателя он адресовал нужный объект или элемент этого объекта. Попытка адресации защищенной области памяти приводит к ошибке выполнения.

Ограничения:

1. Адресная переменная не может быть формальным параметром, именем функции или элементом общего блока. Адресную переменную нельзя инициализировать при ее объявлении или в операторе DATA. Указатель не может появляться в операторе объявления типа и не может быть инициализирован в операторе DATA.

2. Операторы ALLOCATE и DEALLOCATE не могут быть использованы с целочисленными указателями.

Указатель может быть размещен на начало свежей области памяти, выделяемой функцией MALLOC. После использования выделенная память может быть освобождена встроенной подпрограммой FREE.

Пример. Сформировать область памяти, занося в байт по адресу $p0 + k$ натуральное число k (k = 0, 127, 1).

```
byte gk
integer k, size /127/, p0
pointer(p, gk)                    ! Функция MALLOC возвращает начальный адрес
p0 = malloc(size)                 ! выделенной памяти
p = p0                            ! Установим указатель p в начало выделенной
do k = 0, size                    ! памяти
 gk = int(k, kind = 1)            ! В ячейку с адресом p заносится значение gk
 p = p + 1                        ! Переход к следующему байту памяти
end do                            ! Просмотр памяти
print '(10i3)', (gk, p = p0, p0 + 5)  ! 1  2  3  4  5  6
call free(p0)                     ! Подпрограмма FREE освобождает выделенную
end                               ! функцией MALLOC память
```

Целочисленные указатели являются расширением CVF и FPS над стандартами Фортран 90 и 95, и поэтому с ними можно работать при отсутствии директивы $STRICT или опции компилятора /4Ys. В основном целочисленные указатели предназначены для организации доступа к произвольной, доступной из программы области памяти ЭВМ.

Замечание. Целочисленные указатели Фортрана подобны указателям СИ. Порядок передачи целочисленных указателей в СИ-функции и приема указателей СИ в Фортран-процедуре рассмотрен в [1].

3.11. Ссылки и адресаты

Память под переменную может быть выделена на этапе компиляции или в процессе выполнения программы. Переменные, получающие память на этапе компиляции, называются *статическими*. Переменные, получающие память на этапе выполнения программы, называются *динамическими*.

Ссылки - это динамические переменные. Выделение памяти под ссылку выполняется либо при ее размещении оператором ALLOCATE, либо после

ее прикрепления к размещенному адресату. В последнем случае ссылка занимает ту же память, которую занимает и адресат.

3.11.1. Объявление ссылок и адресатов

Для объявления ссылки (переменной с атрибутом POINTER) используется атрибут или оператор POINTER, который, конечно, не следует смешивать с оператором объявления целочисленного указателя. Адресатом может быть:

- прикрепленная ссылка;
- переменная, имеющая атрибут TARGET (объявленная с атрибутом TARGET или в операторе с тем же именем);
- выделенная оператором ALLOCATE свежая область памяти.

Ссылками и адресатами могут быть как скаляры, так и массивы любого встроенного или производного типа.

Пример:

```
! Объявление с использованием атрибутов
integer(4), pointer :: a, b(:), c(:,:)          ! Объявление ссылок
integer(4), target, allocatable :: b2(:)        ! Объявление адресата
integer(4), target :: a2
! Объявление с использованием операторов POINTER и TARGET
integer(4) d, e, d2 /99/
pointer d, e                                    ! Объявление ссылок
target d2                                       ! Объявление адресата
```

3.11.2. Прикрепление ссылки к адресатам

Для прикрепления ссылки к адресату используется оператор =>. После прикрепления ссылки к адресату можно обращаться к адресату, используя имя ссылки. То есть ссылка может быть использована в качестве второго имени (псевдонима) адресата.

Пример:

```
integer, pointer :: a, d, e                 ! Объявление ссылок
integer, pointer, dimension(:) :: b
integer, target, allocatable :: b2(:)       ! Объявление адресатов
integer, target :: a2, d2 = 99
allocate(b2(5))                             ! Выделяем память под массив-адресат
a2 = 7
a => a2                                     ! Прикрепление ссылки к адресату
b2 = (/1, -1, 1, -1, 1/)                    ! Изменение ссылки приводит к
b => b2                                     ! изменению адресата, а изменение адресата
b = (/2, -2, 2, -2, 2/)                     ! приводит к изменению  ссылки
print *, b2                                 !       2       -2       2       -2       2
b2 = (/3, -3, 3, -3, 3/)
print *, b                      !       3       -3       3       -3       3
```

```
a => d2                          ! Теперь ссылка a присоединена к d2
d => d2; e => d2                 ! Несколько ссылок присоединены к одному адресату
print *, a, d, e                 !       99      99      99
a = 100                          ! Изменение a вызовет изменение всех
                                 ! ассоциированных с a объектов
print *, a, d, e, d2             !      100     100     100     100
deallocate(b2)                   ! Освобождаем память
nullify(b)                       ! Обнуление ссылки
end
```

Нельзя прикреплять ссылку к не получившему память адресату, например:

```
integer(4), pointer :: b(:)
integer(4), allocatable, target :: b2(:)
b => b2                          ! Ошибочное прикрепление ссылки
allocate(b2(5))                  ! к неразмещенному адресату
b2 = (/1, -1, 1, -1, 1/)
print *, b
```

Если объект имеет атрибуты TARGET или POINTER, то в ряде случаев такие же атрибуты имеет его подобъект. Так, если атрибут TARGET (POINTER) имеет весь массив, то и сечение массива, занимающее непрерывную область в памяти, и элемент массива обладают атрибутом TARGET (POINTER), например:

```
integer, target :: a(10) = 3
integer, pointer :: b(:), bi
b => a(1:5)                      ! Ссылка и адресат полностью тождественны
! Ошибочен оператор прикрепления ссылки  к нерегулярному сечению
! массива, имеющему атрибут target: b => a(1:5:2)
bi => b(5)                       ! Элемент массива также имеет атрибут POINTER
print *, b                       !       3       3       3       3       3
```

Однако подстрока строки, имеющей атрибуты TARGET или POINTER, этими атрибутами не обладает, например:

```
character(len = 10), target :: st = '1234567890'
character(len = 5), pointer :: ps2
ps2 => st(:5)                    ! Ошибка
```

Если адресатом является ссылка, то происходит прямое копирование ссылки. Поэтому адресатом в операторе прикрепления ссылки может быть произвольное заданное индексным триплетом (разд. 4.5) сечение массива-ссылки. Само же заданное векторным индексом сечение не может быть адресатом ссылки. Например:

```
character(len = 80), pointer :: page(:), part(:), line, st*10
allocate(page(25))
part => page
part => page(5:15:3)          ! Было бы ошибкой, если бы page
```

```
                                ! имел атрибут TARGET
line => page(10)                ! Элемент массива также имеет атрибут POINTER
st => page(2)(20:29)            ! Ошибка: подстрока не обладает атрибутом POINTER
```

Если ссылка-адресат не определена или откреплена, то копируется состояние ссылки. Во всех остальных случаях адресаты и ссылки становятся тождественными (занимают одну и ту же область памяти).

Тип, параметры типа и ранг ссылки в операторе прикрепления ссылки должны быть такими же, как и у адресата. Если ссылка является массивом, то она воспринимает форму адресата. При этом нижняя граница всегда равна единице, а верхняя - размеру экстента массива-адресата. Например:

```
integer, pointer :: a(:)
integer, target :: i, b(10) = (/ (i, i = 1, 10) /)
! Нижняя граница массива a равна единице, верхняя  - трем
a => b(3:10:3)
print *, (a(i), i = 1, size(a))      !       3       6       9
```

Такое свойство ссылок позволяет заменить ссылкой часто применяемое сечение и обращаться к его элементам, используя в каждом его измерении индексы от 1 до *n*, где *n* - число элементов сечения в измерении.

Ссылка может быть в процессе выполнения программы прикреплена поочередно к разным адресатам. Несколько ссылок могут иметь одного адресата. Факт присоединения ссылки к адресату можно обнаружить, применив справочную функцию ASSOCIATED, имеющую синтаксис:

result = ASSOCIATED(*ссылка* [, *адресат*])

Параметр *адресат* является необязательным. Если параметр *адресат* отсутствует, то функция возвращает .TRUE., если ссылка присоединена к какому-либо адресату. Если адресат задан и имеет атрибут TARGET, то функция возвращает .TRUE., если ссылка присоединена к адресату. Функция также вернет .TRUE., если и ссылка и адресат имеют атрибут POINTER и присоединены к одному адресату. Во всех других случаях функция возвращает .FALSE.. Возвращаемое функцией значение имеет стандартный логический тип.

Пример:

```
real, pointer :: c(:), d(:), g(:)
real, target :: e(5)
logical sta
c => e                          ! Прикрепляем ссылки c и d к адресату
d => e
sta = associated(c)             ! В первых трех случаях sta равен .TRUE.
sta = associated(c, e)          ! в последнем - .FALSE.
sta = associated(c, d)
sta = associated(g)
```

3.11.3. Инициализация ссылки. Функция NULL

Ссылку можно инициализировать, применив функцию NULL:

```
real(4), dimension(:), pointer :: pa => null( )
```

Функция NULL дает ссылке статус "не ассоциирована с адресатом". Этот статус позволяет, например, использовать ссылку в качестве фактического параметра до ее прикрепления к адресату, например:

```
program null_test
 real(4), dimension(:), pointer :: pa => null( )
 interface
  subroutine poas(pa)
   real(4), dimension(:), pointer :: pa
  end subroutine poas
 end interface
 call poas(pa)                    ! Параметр - неприкрепленная ссылка
 print *, pa(2)                   !     3.500000
end program null_test

subroutine poas(pa)
 real(4), dimension(:), pointer :: pa
 allocate(pa(5))
 pa = 3.5
end subroutine poas
```

Функция NULL может быть использована и среди исполняемых операторов:

```
pa => null( )
```

3.11.4. Явное открепление ссылки от адресата

Ссылку можно открепить от адресата, используя оператор NULLIFY:

NULLIFY(*pname*)

pname - список ссылочных переменных или компонентов структуры, которые следует открепить от их адресатов. Все элементы списка должны иметь атрибут POINTER. Например:

```
integer, pointer :: a(:), b, c(:, :, :)
 ...
nullify(a, b, c)
```

Кроме открепления от адресата, оператор NULLIFY можно использовать для инициализации (обнуления) ссылки. Не является ошибкой открепление неприкрепленной ссылки.

Оператор NULLIFY не освобождает адресата. Иными словами, если адресатом ссылки является выделенная оператором ALLOCATE память, то после открепления ссылки память не освобождается и остается недоступной, если к памяти была прикреплена лишь одна ссылка.

Недоступная память не образуется, если предварительно память освобождается оператором DEALLOCATE. Например:

```
integer, pointer :: c(:, :, :)
...
allocate(c(2, 5, 10))
...
deallocate(c)
nullify(c)
```

Та же опасность образования недоступной области памяти существует и при переназначении ссылки к другому адресату посредством оператора прикрепления ссылки =>. Например, в следующем фрагменте образуется недоступная память:

```
integer, pointer :: a(:)
integer, target :: b(5)
...
allocate(a(5))                ! Адресатом ссылки a является выделяемая память
...
a => b                        ! Выделенная ранее под ссылку a память становится
...                           ! недоступной для последующего использования
```

В следующем фрагменте переназначение ссылки выполнено без потери памяти:

```
allocate(a(5))
...
deallocate(a)                 ! Освобождение памяти. Память доступна для
a => b                        ! последующего использования
...
```

Заметим, что после выполнения оператора DEALLOCATE состояние ссылки становится неопределенным и может быть определено либо в результате присоединения к другому адресату, либо в результате обнуления в операторе NULLIFY.

3.11.5. Структуры со ссылками на себя

Компонент производного типа может иметь атрибут POINTER и при этом ссылаться на переменную того же или другого производного типа:

```
type entry                    ! Объявление типа entry
  real val
  integer index
  type(entry), pointer :: next    ! Ссылка на объект типа entry
end type entry
```

Это свойство позволяет использовать ссылки для формирования связных списков. Рассмотрим в качестве примера однонаправленный список, представляющий структуру данных, каждый элемент которой содержит две

части - *поля с данными* и *поле с адресом* следующего элемента данных списка. Адрес, используемый для доступа к следующему элементу, называется *указателем*. Первый элемент списка называется *вершиной списка*. Последний элемент списка содержит нулевой указатель. Однонаправленный список с записями одинаковой длины можно представить в виде рис. 3.2.

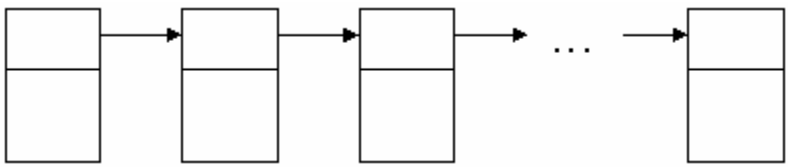


Рис. 3.2. Схема однонаправленного списка с записями одинаковой длины

Разработаем программу формирования и редактирования однонаправленного списка. Пусть помимо поля с указателем каждый элемент списка содержит еще два поля, одно из которых (*index*) используется для индексирования элементов списка. Редактирование состоит из двух операций: добавления и удаления элемента из однонаправленного списка. Схема добавления элемента в однонаправленный список приведена на рис. 3.3, а удаления - на рис. 3.4.

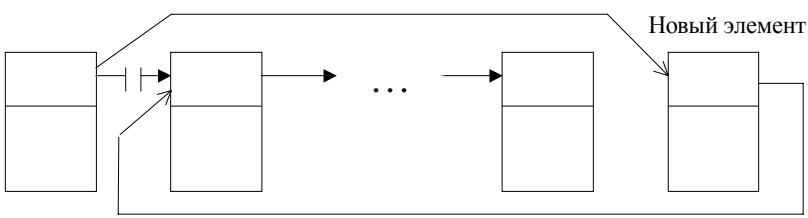


Рис. 3.3. Схема включения элемента в однонаправленный список

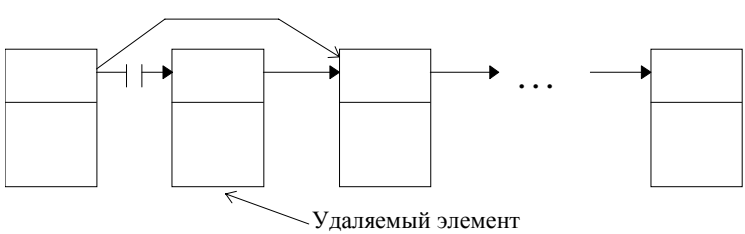


Рис. 3.4. Схема удаления элемента из однонаправленного списка

По приведенным рисункам легко записать алгоритмы формирования списка и его модификации.

Алгоритм формирования списка из *n* элементов (оформлен в виде внутренней подпрограммы *newtree*):

1°. Начало (список формируется начиная с последнего элемента).
2°. Обнулить адрес последнего элемента списка *tree* - NULLIFY(*tree*).
3°. С параметром *k* = 1, *n* выполнить:
    Выделить память под элемент *top*.
    В поле с адресом элемента *top* занести адрес элемента,
    добавленного на шаге *k* - 1, т. е. адрес вершины *tree*.
    (Таким образом, каждый вновь добавляемый элемент (кроме
    последнего элемента списка) будет указывать на предыдущий.)
    Переместить вершину списка на последний добавленный элемент,
    выполнив *tree* => *top*.
  конец цикла 3°.
4°. Конец.

Приведем далее текст программы, которая позволяет формировать список из заданного числа элементов, выполнять его просмотр и редактирование.

```
module eni                          ! Модуль описания типа entry
 type entry
   real val
   integer index
   type(entry), pointer :: next     ! Ссылка, которую можно присоединить
 end type entry                     ! к переменной типа entry
end module eni

program one_d_li                    ! Формирование, просмотр
use eni                             ! и редактирование списка
type(entry), pointer :: tree, top, current, newtop
call newtree( )                     ! Список сформирован; top ссылается
                                    ! на последний элемент списка
call wali(top)                      ! Просмотр списка
tree => top                         ! Перемещение в вершину списка
call chava(5)                       ! Изменим поле val у двух соседних
call wali(top)                      ! Просмотр списка
call delem(7)                       ! Исключим из списка элемент с index = 7
call wali(top)                      ! Просмотр списка
tree => top

call insel(700.0, 8)                ! Вставим элемент после элемента с index = 8
call wali(top)                      ! Просмотр списка

contains

 subroutine newtree( )              ! Создание списка
  integer i
  nullify(tree)                     ! Обнуляем конец списка. В конце списка
                                    ! ASSOCIATED(tree) будет возвращать .FALSE.
  do i = 1, 10                      ! Цикл формирования списка
  ! Размещаем новый элемент списка и заносим информацию
  ! в поля с данными
```

```
  allocate(top)                  ! Ссылка top содержит в качестве
  top = entry(11.0*i, i, tree)   ! компонента ссылку tree
  tree => top                    ! Установим указатель списка на
 end do                          ! вновь добавленный элемент
end subroutine newtree
subroutine wali(top)             ! Просмотр списка начиная с элемента top
 type(entry), pointer :: top
 tree => top                     ! Прикрепляем ссылку tree к top
 do while(associated(tree))      ! Цикл просмотра списка
  print *, tree%val, tree%index
  current => tree%next           ! Переход к следующему элементу
  tree => current
 end do
 read *                          ! Ожидаем нажатия Enter
end subroutine wali
subroutine delem(ind)            ! Удаление элемента из списка
 integer ind
 tree => top
 do while(associated(tree))
  if(tree%index == ind) then     ! Следующий элемент имеет index = 7
  ! Присоединим указатель элемента с index = in к указателю элемента
  ! с index = in + 2, выполнив тем самым исключение элемента с index = in + 1
   if(associated(tree%next)) then
    current => tree%next
    tree%next => tree%next%next
    deallocate(current)          ! Освобождаем память после исключения
   end if                        ! элемента из списка
   exit
  end if
  current => tree%next
  tree => current
 end do
end subroutine delem
subroutine insel(val, ind)       ! Добавляет элемент в список tree
 integer ind
 real val
 do while(associated(tree))
  if(tree%index == ind) then
   allocate(newtop)
   newtop = entry(val, ind - 1, tree%next)
   tree%next => newtop
   if(tree%index > 1) then
    newtop%next => tree%next%next
   else                          ! При вставке вслед за последним
    nullify(newtop%next)         ! элементом обнуляем ссылку
   end if
```

```
    exit
   end if
   current => tree%next
   tree => current
  end do
 end subroutine insel
 subroutine chava(ind)              ! Изменим теперь значение поля элементов
  integer ind                       ! списка с index = ind и index = ind - 1
  do while(associated(tree))
   if(tree%index == ind) then
    tree%val = 500                  ! Поле val элемента списка с index = ind
    tree%next%val = 400             ! Поле val элемента списка с index = ind - 1
    exit
   end if
   current => tree%next
   tree => current
  end do
 end subroutine chava
end program one_d_li
```

Замечания:

1. В общем случае элементы списка располагаются в свободной памяти ЭВМ произвольным образом. Поэтому подобные списки иногда называют разреженными векторами.

2. Используя ссылки в качестве компонентов производного типа, можно сформировать циклические, двунаправленные списки и более сложные структуры данных.

3.11.6. Ссылки как параметры процедур

Ссылка может быть формальным параметром процедуры. В этом случае фактический параметр также должен быть ссылкой. Ранги фактического и формального параметров-ссылок должны совпадать. При вызове процедуры состояние привязки фактического параметра передается формальному, и, наоборот, при завершении работы процедуры состояние привязки формального параметра передается фактическому. Однако при выходе адресат может стать неопределенным, если, например, в процессе работы процедуры ссылка была прикреплена к локальной переменной, не имеющей атрибут SAVE.

В случае модульной или внутренней процедуры компилятор знает, является ли формальный параметр ссылкой или нет. В случае внешней процедуры компилятору необходимо явно указать, что формальный параметр является ссылкой. Это достигается за счет задания явного интерфейса. Для формального параметра-ссылки недопустимо задание атрибута INTENT, устанавливающего вид связи параметра.

Пример:

```
real, pointer, dimension(:) :: a, a2
interface
 subroutine pab(b, b2)
  real, pointer :: b(:), b2(:)
 end
end interface                          ! При вызове процедуры состояние привязки
call pab(a, a2)                        ! формального параметра передается фактическому
print '(10f5.1)', a                        ! 1.0  2.03.0 4.05.0
print '(10f5.1)', a2                       !  .0   .0.0      .0.0
print *, associated(a), associated(a2)     ! T T
end

subroutine pab(b, b2)
 real, pointer, dimension(:) :: b, b2
 integer k
 real, target, save :: c(5)
 real, target, automatic :: c2(5)
 c  = (/ (1.0*k, k = 1, 5) /)
 c2 = (/ (1.0*k, k = 1, 5) /)
 b => c                                ! После выхода из-за отсутствия у c2 атрибута
 b2 => c2                              ! SAVE адресат ссылки a2 будет неопределен
end subroutine pab
```

Если фактическому параметру-ссылке соответствует формальный параметр, не являющийся ссылкой, то ссылка-параметр должна быть прикреплена к адресату, который и будет ассоциирован с формальным параметром.

Пример:

```
integer, pointer :: a(:)
integer, target :: c(5) = 2
a => c                                 ! Адресат с фактического параметра-ссылки a
call pab(a, size(a))                   ! ассоциируется с не являющимся ссылкой
                                       ! формальным параметром b подпрограммы pab
print *, a                             !     7     7     7     7     7
print *, c                             !     7     7     7     7     7
end

subroutine pab(b, n)
integer n, b(n)
 b = b + 5
end subroutine pab
```

3.11.7. Параметры с атрибутом TARGET

Фактический параметр процедуры может иметь атрибут TARGET. В этом случае ссылки, прикрепленные к такому параметру, не прикрепляются к соответствующему формальному параметру, а остаются

связанными с фактическим параметром. Если же формальный параметр имеет атрибут TARGET, то любая прикрепленная к нему ссылка (если только она явно не откреплена от адресата) остается неопределенной при выходе из процедуры. К процедуре, формальный параметр которой имеет атрибут TARGET, должен быть организован явный интерфейс. Таким интерфейсом обладают внутренние и модульные процедуры, обретают его и внешние процедуры после их описания в интерфейсном блоке, расположенном в вызывающей программной единице (разд. 8.11.3).

3.11.8. Ссылки как результат функции

Функция может иметь атрибут POINTER. В этом случае результатом функции является ссылка и к ней можно прикрепить другую ссылку. Такое использование функции целесообразно, если, например, размер результата зависит от вычислений в самой функции.

При входе в функцию вначале ссылка-результат не определена. Внутри функции она должна быть прикреплена к адресату или определена посредством оператора NULLIFY как открепленная.

Обращение к функции-ссылке можно выполнить из выражения. В этом случае адресат ссылки-результата должен быть определен и его значение будет использовано при вычислении выражения. Также функция-ссылка может быть ссылочным компонентом конструктора структуры.

Тема "ссылки" тесно связана с темой "массивы", поэтому дальнейшее рассмотрение ссылок мы отложим до следующей главы.

Пример. Получить массив *a* из неотрицательных чисел массива *c*.

```
integer, pointer :: a(:)
integer :: c(10) = (/ 1, -2, 2, -2, 3, -2, 4, -2, 5, -2 /)
a => emi(c)                          ! Прикрепляем ссылку к результату функции
print *, a                           !       1       2       3       4       5
! Использование функции-ссылки в выражении
print *, 2 * emi(c)                  !       2       4       6       8      10

 contains                            ! Внутренняя процедура обладает
  function emi(c)                    ! явно заданным интерфейсом
   integer, pointer :: emi(:)        ! Результатом функции является ссылка
   integer c(:), i, k                ! c - перенимающий форму массив
   k = count(c >= 0)                 ! k - число неотрицательных элементов
   allocate(emi(k))                  ! Прикрепляем ссылку к адресату
   emi = pack(c, mask = c >= 0)      ! Заносим в ссылку неотрицательные
  end function emi                   ! элементы массива c
end
```

Замечания:

1. Если функцию *emi* оформить как внешнюю, то в вызывающей программе потребуется блок с интерфейсом к этой функции, поскольку, во-первых, она

возвращает ссылку, а, во-вторых, ее формальным параметром является перенимающий форму массив.

2. Встроенные функции COUNT и PACK приведены в разд. 4.12.1 и 4.12.4.2.

4. Массивы

Массив - это именованный набор из конечного числа объектов одного типа. Объектами (элементами) массива могут быть данные как базовых, так и производных типов. Используя атрибут PARAMETER, можно задать массив-константу. В отличие от простых переменных, предназначенных для хранения отдельных значений, массив является *составной* переменной. Также к составным относятся объекты символьного и производного типов.

Массивы, так же как и объекты производного типа, обеспечивают доступ к некоторому множеству данных при помощи одного имени, которое называется *именем массива*. Также имя массива используется для обеспечения доступа к элементу или группе элементов (сечению) массива.

Массивы могут быть *статическими* и *динамическими*. Под статические массивы на этапе компиляции выделяется заданный объем памяти, которая занимается массивом во все время существования программы. Память под динамические массивы выделяется в процессе работы программы и при необходимости может быть изменена или освобождена. К динамическим массивам относятся массивы-ссылки, размещаемые и автоматические массивы. Последние могут появляться только в процедурах.

Память под массивы-ссылки выделяется либо в результате выполнения оператора ALLOCATE, либо после прикрепления ссылки к уже размещенному объекту-адресату. Размещаемые массивы получают память только после выполнения оператора ALLOCATE.

4.1. Объявление массива

Массив характеризуется числом измерений, которых может быть не более семи. Число измерений массива называется его *рангом*. Число элементов массива называется его *размером*. Также массив характеризуется *формой*, которая определяется его рангом и *протяженностью* (*экстентом*) массива вдоль каждого измерения.

Оператор

```
real b(2, 3, 10)
```

объявляет массив *b* ранга 3. Размер массива равен 2*3*10 = 60. Форма массива - (2, 3, 10).

Каждая размерность массива может быть задана *нижней* и *верхней* границей, которые разделяются двоеточием, например:

```
real c(4:5, -1:1, 0:9)
```

Ранг, форма и размер массивов *b* и *c* совпадают. Такие массивы называются *согласованными*.

Нижняя граница и последующее двоеточие при объявлении массива могут быть опущены, тогда по умолчанию нижняя граница принимается

равной единице. Объявление массива выполняется при объявлении типа либо операторами DIMENSION, ALLOCATABLE и POINTER. Также массив можно объявить в операторе COMMON. Приведем различные способы объявления статического одномерного массива целого типа из 10 элементов.

Можно использовать оператор объявления типа:

```
integer a(10)                    ! или a(1:10)
```

Зададим границы в виде константного выражения (что рекомендуется):

```
integer, parameter :: n = 10
integer a(1:n)
```

Используем теперь атрибут DIMENSION:

```
integer, dimension(10) :: a
```

а затем оператор DIMENSION:

```
integer a
dimension a(10)
```

Приведенные объявления статического одномерного массива определяют массив *a* из 10 объектов (элементов) с именами *a*(1), *a*(2), ..., *a*(10). Схема расположения элементов массива *a* в памяти компьютера приведена на рис. 4.1.

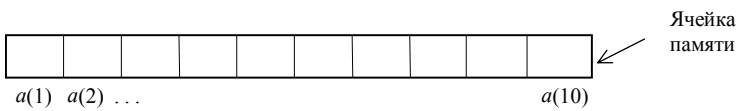


Рис. 4.1. Расположение элементов массива в памяти ЭВМ

Запись *a*(*i*) отсылает нас к *i*-му элементу массива *a*. Переменную *i* называют *индексной переменной* или просто *индексом*.

Динамический одномерный массив можно объявить, применяя операторы ALLOCATABLE или POINTER:

```
real a, b
allocatable a(:)                 ! Измерения динамического массива
pointer b(:)                     ! задаются двоеточием (:)
```

или атрибуты ALLOCATABLE или POINTER:

```
real, allocatable :: a(:)
real, pointer :: b(:)
```

При объявлении статического массива может быть выполнена его инициализация:

```
integer a(10) /1, 2, 3, 4, 4, 4, 5, 5, 5, 5/
```

или с использованием коэффициента повторения:

```
integer a(10) /1, 2, 3, 3*4, 4*5/
```

или с использованием оператора DATA:

```
integer a(10)
data a /1, 2, 3, 3*4, 4*5/
```

или с использованием конструктора массива:

```
integer :: a(10) = (/ 1, 2, 3, 4, 4, 4, 5, 5, 5, 5 /)
```

или с использованием в конструкторе массива циклического списка:

```
integer :: i, j, a(10) = (/ 1, 2, 3, (4, i = 4, 6), (5, i = 7, 10) /)
```

Во всех приведенных примерах после инициализации массива *a*

```
a(1) = 1, a(2) = 2, a(3) = 3, a(4) = 4, a(5) = 4, a(6) = 4, a(7) = 5, a(8) = 5, a(9) = 5,
a(10) = 5.
```

При инициализации необходимо, чтобы число констант в списке значений равнялось числу элементов массива. Используя в операторе DATA неявный цикл, можно инициализировать часть массива.

Одним оператором может быть выполнено объявление более одного массива. Например, оператор

```
real b(-3:3) /7*1.0/, g2(6:8)        ! Массив g2 не определен
```

объявляет два одномерных массива - массив *b* из семи элементов с именами *b*(-3), *b*(-2), ..., *b*(3) и массив *g*2 из трех элементов с именами *g*2(6), *g*2(7), *g*2(8). Все элементы массива *b* равны 1.0.

Пример. Найти сумму элементов одномерного массива.

```
real b(-3:5) /1.1, 2.2, 3.3, 4.4, 5.5, 4*7.8/, s
s = 0.0
do k = -3, 5
  s = s + b(k)
end do
write(*, *) ' s = ', s                ! s = 47.7
! Та же задача решается с применением встроенной функции SUM:
write(*, *) ' s = ', sum(b)           ! s = 47.7
```

Аналогично выполняется объявление двумерного массива:

```
integer b(2, 4)                       ! Статический массив из восьми элементов
real, pointer :: c(:, :)              ! Динамический массив-ссылка
```

Первое объявление определяет двумерный массив *b* из восьми элементов с именами *b*(1,1), *b*(2,1), *b*(1,2), *b*(2,2), *b*(1,3), *b*(2,3), *b*(1,4), *b*(2,4). Двумерный массив *b*(1:2, 1:4) можно представить в виде таблицы (рис. 4.2), содержащей 2 строки и 4 столбца.

		j			
		1	2	3	4
i	1	*b*(1, 1)	*b*(1, 2)	*b*(1, 3)	*b*(1, 4)
	2	*b*(2, 1)	*b*(2, 2)	*b*(2, 3)	*b*(2, 4)

Рис. 4.2. Представление двумерного массива в виде таблицы

Память компьютера является одномерной, поэтому элементы двумерного массива *b* расположены в памяти в линейку так, как это показано на рис. 4.3.

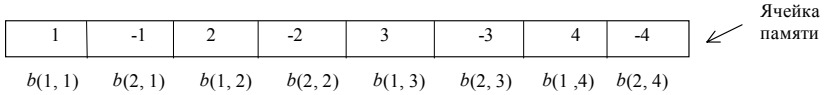


Рис. 4.3. Расположение элементов двумерного массива в памяти ЭВМ

Запись *b*(*i*, *j*) отсылает нас к *j*-му элементу массива в его *i*-й строке, где *i* и *j* - индексы массива *b*.

Во многих языках программирования, например в СИ, элементы двумерного массива располагаются в памяти ЭВМ по строкам, в Фортране – по столбцам, т. е. быстрее изменяется первый индекс массива. В более общем случае в Фортране для многомерного массива при размещении его элементов в памяти ЭВМ закономерность изменения индексов можно отобразить вложенным циклом (на примере трехмерного массива *a*(2, 4, 6):

```
do k = 1, 6
  do j = 1, 4
    do i = 1, 2                    ! Быстрее всего изменяется индекс i
      размещение в памяти элемента с именем a(i, j, k)
    end do
  end do
end do
```

Иными словами, при размещении многомерного массива в памяти ЭВМ быстрее всего изменяется самый левый индекс массива, затем следующий за ним индекс и т. д.

Это обстоятельство следует учитывать при В/В и инициализации многомерного массива. В случае двумерного массива перечисление значений приведет к инициализации по столбцам (соответствие элемент - значение показано на рис. 4.3):

```
integer b(2, 4) / 1, -1, 2, -2, 3, -3, 4, -4 /
```

Инициализация по строкам может быть выполнена оператором DATA:

```
data ((b(i, j), j = 1, 4), i = 1, 2) /1, -1, 2, -2, 3, -3, 4, -4/
```

или в конструкторе массива при надлежащем применении функции RESHAPE:

```
integer :: b(2, 4) = reshape((/                              &
                        1,   -1    2   -2                    &
                        3    -3    4   -4 /), shape = (/ 2, 4 /), order = (/ 2, 1 /))
```

Пример. Найти произведение положительных элементов двумерного массива.

```
real  b(-2:1, 6:8) /1.1, 2.2, 3.3, 4.4, 5.5, 7*-1.1/, p
integer i, j
p = 1.0
do j = 6, 8
 do i = -2, 1
  if(b(i, j) > 0) p = p * b(i, j)
 end do
end do
write(*, *) ' p = ', p                  ! p = 193.2612
! Для вычисления произведения можно применить встроенную функцию PRODUCT:
p = product(b, mask = b > 0.0)
write(*, *) ' p = ', p                  ! p = 193.2612
```

Замечание. Порядок размещения элементов многомерного массива следует учитывать при организации вложенных циклов. Так, цикл

```
do j = 6, 8
 do i = -2, 1                           ! Перебираем элементы очередного столбца массива
  if(b(i, j) > 0) p = p * b(i, j)
 end do
end do
```

отработает быстрее, чем цикл

```
do i = -2, 1
 do j = 6, 8                            ! Неестественный порядок перебора элементов
  if(b(i, j) > 0) p = p * b(i, j)       ! массива b
 end do
end do
```

Это объясняется тем, что в первом случае во внутреннем цикле обеспечивается естественная и поэтому более производительная последовательность доступа к элементам массива.

Индексом массива может быть целочисленное выражение:

```
real :: b(5, 10) = 5.1
real :: a(5, 5), c(30), r = 7.0
c(int(r)*2 + 1) = 2.0                   ! Индекс массива - целочисленное выражение
a(1, 2) = b(int(c(15)), int(sqrt(r)))
write(*, *) a(1, 2), b(2, 2)            !       5.100000       5.10000
```

При объявлении массива помимо задающих динамические массивы атрибутов ALLOCATABLE и POINTER и атрибута задания формы массива DIMENSION могут быть использованы атрибуты INTENT, OPTIONAL, PARAMETER, PRIVATE, PUBLIC, SAVE и TARGET.

4.2. Массивы нулевого размера

Массив, как, впрочем, и строка, может иметь нулевой размер. Всегда, когда нижняя граница превосходит соответствующую верхнюю границу, массив имеет размер 0. Например:

```
real b2(0, 15), d(5, 20, 1:-1)
print *, size(b2), size(d)          !       0       0
```

Массивы нулевого размера всегда считаются определенными и при использовании подчиняются обычным правилам.

4.3. Одновременное объявление объектов разной формы

При необходимости можно одним оператором объявлять объекты разной формы. Так, оператор

```
real, dimension(10) :: a, b, b2(15), d(5, 20)
```

объявляет массивы *a* и *b* формы (10), массив *b*2 формы (15) и массив *d* формы (5, 20). То есть приоритетом при использовании атрибута DIMENSION обладает явное описание формы массива.

Оператор

```
integer na, nb, a(10), d(5, 20)
```

объявляет скаляры *na* и *nb* и массивы *a* и *d* разной формы.

4.4. Элементы массива

В этом разделе мы обобщим понятие элемента массива. Массив может содержать элементы встроенных и производных типов данных:

```
type order                                  ! Описание заказа
 integer ordnum, cus_id
 character(15)  item(10)
end type
! Примеры массивов символьного и производного типа
character(20) st(10) /10*'T-strings'/       ! st -  массив строк
type(order) cord, orders(100)               ! orders - массив заказов
```

Приведенные описания определяют массивы *st*, *orders*, *cord%item* и *orders*(*k*)%*item* (*k* - целое и $0 \le k \le 100$). Элементы массивов являются скалярами.

Примеры элементов массивов:

```
st(7),  orders(15),  cord%item(7),  orders(10)%item(8)
```

Из символьного массива можно извлечь еще один объект данных - подстроку, например:

```
print *, st(7)(1:6)                    ! T-stri
```

или

```
character(15) itord
orders = order( 2000, 455, (/ ('Item', k = 1, 10) /) )
itord = orders(10)%item(8)
print *, itord(3:4)                    ! em
```

Однако содержащаяся в символьном массиве подстрока по соглашению не рассматривается как элемент массива.

В общем случае элемент массива - это скаляр вида

частный указатель [%*частный указатель*...]

где частный указатель есть

частное имя [(*список индексов*)]

Если частный указатель является именем массива, то он обязан иметь список индексов, например *orders*(10)%*item*(8). Число индексов в каждом *списке индексов* должно равняться рангу массива или массива - компонента производного типа. Каждый индекс должен быть целым скалярным выражением, значение которого лежит в пределах соответствующих границ массива или массива-компонента.

4.5. Сечение массива

Получить доступ можно не только к отдельному элементу массива, но и к некоторому подмножеству его элементов. Такое подмножество элементов массива называется *сечением массива*. Сечение массива может быть получено в результате применения *индексного триплета* или *векторного индекса*, которые при задании сечения подставляются вместо одного из индексов массива.

Индексный триплет имеет вид:

[*нижняя граница*] : [*верхняя граница*] [:*шаг*]

Каждый из параметров триплета является целочисленным выражением. *Шаг* изменения индексов может быть и положительным и отрицательным, но не может равняться нулю. Все параметры триплета являются необязательными.

Индексный триплет задает последовательность индексов, в которой первый элемент равен его нижней границе, а каждый последующий больше (меньше) предыдущего на величину шага. В последовательности находятся все задаваемые таким правилом значения индекса, лежащие между границами триплета. Если же нижняя граница больше верхней и шаг

положителен или нижняя граница меньше верхней и шаг отрицателен, то последовательность является пустой.

Пример:

```
real a(1:10)
a(3:7:2) = 3.0          ! Триплет задает сечение массива с элементами
                        ! a(3), a(5), a(7), которые получат значение 3.0
a(7:3:-2) = 3.0         ! Элементы a(7), a(5), a(3) получат значение 3.0
```

При отсутствии нижней (верхней) границы триплета ее значение принимается равным значению нижней (верхней) границы соответствующего экстента массива. Так, *a*(1:5:2) и *a*(:5:2) задают одно и то же сечение массива *a*, состоящее из элементов *a*(1), *a*(3), *a*(5). Сечение из элементов *a*(2), *a*(5), *a*(8) может быть задано так: *a*(2:10:3) или *a*(2::3).

Пример:

```
real a(10), r /4.5/
a(2::3) = 4.0           ! Элементы a(2), a(5), a(8) получат значение 4.0
a(7:9) = 5.0            ! Элементы a(7), a(8), a(9) получат значение 5.0
a(:) = 3.0              ! Все элементы массива получат значение 3.0
a(::3) = 3.0            ! Сечение из элементов a(1), a(4), a(7), a(10)
a(::int(r / 2)) = 3.0   ! Параметр триплекса - целочисленное выражение
print *, size(a(4:3))   !     0     (сечение нулевого размера)
                        ! Функция SIZE возвращает размер массива
```

Нижняя граница триплета не может быть меньше нижней границы соответствующего экстента массива. Так, ошибочно задание сечения *a*(-2:5:3) в массиве *a*(1:10). Верхняя граница триплета должна быть такой, чтобы последний элемент задаваемой триплетом последовательности не превышал верхней границы соответствующего экстента массива. Например, в массиве *a*(1:10) допустимо сечение *a*(3:12:5). Верхняя граница триплета всегда должна присутствовать при использовании триплета в последней размерности перенимающего размер массива (разд. 4.9.3).

В случае многомерного массива сечение может быть задано посредством подстановки индексного триплета (впрочем, так же, как и векторного индекса) вместо одного или нескольких индексов, например:

```
real a(8, 3, 5)
a(1:4:2, 2:3, 4) = 4.0
```

В заданном сечении в первом измерении индекс может принимать значения 1 и 3, во втором - 2 и 3, а в третьем - только 4. Таким образом, сечение обеспечивает доступ к элементам *a*(1, 2, 4), *a*(1, 3, 4), *a*(2, 2, 4) и *a*(2, 3, 4). Поскольку в третьем измерении сечения индекс фиксирован, то сечение является двумерным массивом с формой (2, 2).

Частными случаями сечений двумерного массива являются его строки и столбцы, например:

```
integer a(5, 8)
a(3, :) = 3                    ! Сечение - третья строка матрицы
a(:, 4) = 7                    ! Сечение - четвертый столбец матрицы
```

Пример. В какой строке матрицы чаще встречается заданное число *k*.

```
integer, parameter :: m = 3, n = 5
integer :: a(m, n), b(m), i, k = 2, km(1)
 a = reshape((/      1,   2,   2,   4,   3,          &
                     2,   4,   2,   8,   2,          &
                    -2,   2,   6,   2,    2 /), shape = (/m, n/), order = (/2, 1/))
do i = 1, m                    ! Запишем число вхождений k в строку i в
 b(i) = count(a(i, :) == k)    ! массив b. Сечение a(i, :) является
end do                         ! i-й строкой массива
km = maxloc(b)
print *, 'Строки в которых k = ', k, ' входит наибольшее число раз'
do i = 1, m
 if(b(i) == b(km(1))) print *, 'Строка ', i
end do
end
```

Замечание. Массив *b* можно сформировать, не применяя цикла:

```
b = count(a == k, 2)           ! Функция COUNT рассмотрена в разд. 4.12.1.
```

Векторный индекс является одномерным целочисленным массивом, содержащим значения индексов, попадающих в сечение исходного массива, например:

```
real a(20), b(10, 10)
integer :: vi(3), vj(2)        ! vi, vj - целочисленные массивы;
vi= (/1, 5, 7/)                ! используются как векторные индексы
vj= (/2, 7/)                   ! для задания сечений массивов a и b
a(vi) = 3.0                    ! a(1), a(5), a(7) получат значение 3.0
b(2, vj) = 4.0                 ! b(2, 2), b(2, 7) - значение 4.0
b(vi, vj) = 5.0                ! b(1, 2), b(1, 7), b(5, 2), b(5, 7),
                               ! b(7, 2) и b(7, 2) - значение 5.0
```

Векторный индекс в отличие от индексного триплета позволяет извлечь в сечение произвольное подмножество элементов массива. Значения индексов должны находиться в пределах границ соответствующей размерности исходного массива. Значения индексов в векторном индексе могут располагаться в произвольном порядке и могут повторяться. Например:

```
real a(10, 8) /80 * 3.0/, b(5)
b = a(3, (/5, 3, 2, 7, 2/))
```

В массив *b* попадут значения элементов сечения массива *a*: *a*(3, 5), *a*(3, 3), *a*(3, 2), *a*(3, 7) и вновь *a*(3, 5).

Сечения с повторяющимися значениями индексов не могут появляться в правой части оператора присваивания и в списке ввода оператора READ. Например, присваивание

```
real a(10)
a(/5, 3, 2, 7, 2/) = (/1.2 , 1.3, 1.4, 1.5, -1.6/)
```

недопустимо, поскольку 1.4 и -1.6 не могут одновременно храниться в *a*(2).

Размер сечения равен нулю, если векторный индекс имеет нулевой размер.

Сечение массива с векторным индексом не может быть внутренним файлом, адресатом ссылки. Если сечение массива с векторным индексом является фактическим параметром процедуры, то оно рассматривается как выражение и соответствующий формальный параметр должен иметь вид связи INTENT(IN). При использовании заданного векторным индексом сечения в качестве фактического параметра в процедуре создается копия этого сечения, которую и адресует соответствующий формальный параметр.

Сечение массива (заданное индексным триплетом или векторным индексом) сохраняет большинство свойств массива и может быть, в частности, использовано в качестве параметра встроенных функций для массивов, элементных и справочных функций и пользовательских процедур.

Пример. Найти сумму положительных элементов главной диагонали квадратной матрицы.

```
integer, parameter :: n = 10
integer :: a(n, n) /100 * 3.0/, i
integer :: b(n * n)
integer :: v(n) = (/ (i + n * (i - 1), i = 1, n) /)
a(9, 9) = -1                          ! v - векторный индекс, содержащий
b=reshape(a, shape = (/100/))         ! номера элементов главной диагонали
                                      ! массива a; b(v) - сечение массива b
print *, sum(b(v), mask=b(v)>0)       !       27
```

Использование сечений позволяет более эффективно решать задачи, для которых раньше применялись DO-циклы (разд. 7.5).

Пример. Поменять порядок следования элементов массива.

```
integer :: i, a(10) = (/ (i, i = 1, 10) /)
a = a(10:1:-1)
print '(10i3)', a                    !  10   9   8   7   6   5   4   3   2   1
```

Сечение, помимо рассмотренных случаев, может быть взято у массивов производных типов, а также содержать массивы - компоненты структур и подстроки символьных массивов. Общий вид сечения массива:

частный указатель [%*частный указатель* ...] ... [(*диапазон подстроки*)]

где частный указатель есть

частное имя [(*список индексов сечения*)]

Число индексов сечения в каждом списке должно равняться рангу массива или массива - компонента структуры. Каждый *индекс сечения* должен быть либо *индексом*, либо *индексным триплетом*, либо *векторным индексом*, например:

```
real a(8, 5, 5)
a(4, 1:4:2, (/2, 5/)) = 4.0          ! 4 - индекс; 1:4:2 - индексный триплет;
                                     ! (/2, 5/)) - векторный индекс
```

Частный указатель ненулевого ранга определяет ранг и форму сечения. Размер сечения равен нулю, если хотя бы один из экстентов частного указателя равен нулю.

Диапазон подстроки может присутствовать, только если последний частный указатель относится к символьному типу и является скаляром или имеет *список индексов сечения*.

Пример сечения, состоящего из подстрок массива:

```
character(len = 20) st(10) /10*'Test String'/
print *, st((/1, 3, 10/))(5:8)       ! Str Str Str
print *, st(2:6:2)(5:8)              ! Str Str Str
```

Сечение массива, имя которого заканчивается именем компонента структуры, также является *компонентом структуры*.

Пример сечений, содержащих массивы - компоненты структур:

```
type order                           ! Описание заказа:
 integer(4) ordnum, cus_id           ! номер заказа, код покупателя
 character(15)  item(10)             ! список вещей заказа
end type
type(order) cord, ords(100)          ! ords - массив заказов
cord%item(2:6) = 'dress'
ords(5:7:2)%item(7) = 'tie'
ords(9)%item((/1, 8, 9/)) = 'blazer'
ords(9)%item((/8, 9/))(8:9) = '20'
print *, cord%item(3), ords(5)%item(7)       ! dress tie
print *, ords(9)%item(1), ords(9)%item(8)    ! blazer blazer 20
```

Образуемое из массивов - компонентов структур сечение не может содержать более одного нескалярного объекта. Так, попытка создать сечение вида

```
ords(5:7:2)%item((/ 1, 8, 9 /)) = 'none'
```

вызовет ошибку компиляции.

Частное имя справа от частного указателя не должно иметь атрибут POINTER. Так, нельзя задать сечение *list*(1:15:2)%*ival* в таком примере:

```
type wip
 real val
 integer, pointer :: ival
end type wip
type(wip) elem, list(20)
allocate(elem%ival)                 ! Правильно
allocate(list(5)%ival)              ! Правильно
allocate(list(1:15:2)%ival)         ! Ошибка
```

4.6. Присваивание массивов

Как было показано выше, массив может быть определен при инициализации в операторах объявления типа или в операторе DATA. Также значения элементов массива можно изменить, присвоив массиву или его сечению результат выражения. Операндом такого выражения может быть *конструктор массива*. Например:

```
real b(5), pi /3.141593/
integer a(5)
b = tan(pi / 4)                     ! Присваивание значения выражения всему массиву
b(3) = -1.0                         ! Присваивание значения третьему элементу массива
write(*,'(7f5.1)') b                ! 1.0  1.0  -1.0  1.0  1.0
a = 2 * (/ 1, 2, 3, 4, 5 /)         ! Конструктор массива как операнд выражения
write(*, *) a                       !    2    4    6    8    10
```

Конструктор массива задает одномерный массив и имеет вид:

(/ *список-значений* /)

Пробелы между круглой скобкой и слешем *не допускаются*.

Список-значений может содержать последовательность скаляров, неявных циклов и массивов любого ранга. Значения в списке разделяются запятыми и должны иметь одинаковый тип и разновидность типа. Каждое значение списка может быть результатом выражения.

Неявный цикл конструктора массива имеет вид:

(*выражение* | *неявный цикл*, *dovar* = *start*, *stop* [, *inc*])

dovar - целочисленная скалярная переменная (параметр цикла).

start, *stop*, *inc* - целочисленные константные выражения, определяющие диапазон и шаг изменения *dovar*. Если *inc* отсутствует, то шаг устанавливается равным единице.

Неявный цикл добавляет в список значений

MAX(*stop* - *start* + INT(*inc* / *inc*), 0)

элементов. Выражение может содержать *dovar*. Возможна организация вложенных неявных циклов.

Если в списке появляется многомерный массив, то его значения берутся в порядке их размещения в памяти ЭВМ. Конструктор массива позволяет сгенерировать значения одномерного массива. При задании значений

многомерного массива следует получить при помощи конструктора одномерный массив необходимого размера, а затем применить функцию RESHAPE и вписать данные в заданную форму. Число элементов в *списке-значений* должно совпадать с размером массива.

Пример 1. Элементы списка - массивы и простой скаляр.

```
integer b(7), c(2, 3), i, j
integer a(3) / 3, 2, 1 /
b = (/ a, a, mod(a(1), 2) /)          ! В списке одномерный массив и скаляр
write(*,'(10i3)') b                   !   3   2   1   3   2   1   1
data ((c(i, j), j = 1, 3), i = 1, 2) /3*1, 3*2/
b = (/ c, -1 /)                       ! В списке двумерный массив и скаляр
write(*, '(10i3)') b                  !   1   2   1   2   1   2  -1
```

Пример 2. Элементы списка - неявные циклы.

```
integer a(5), i, k
real :: r(7)
real, parameter :: pi = 3.141593
logical fl(10)
a = (/ (i, i = 1, 5) /)
write(*, *) a                         !   1   2   3   4   5
r = (/(cos(real(k) * pi / 180.0), k = 1, 14, 2)/)
write(*, '(10f5.1)') r                !  1.0   1.0   1.0   1.0   1.0   1.0   1.0
fl = (/(.true., k = 1, 5), (.false., k = 6, 10)/)
write(*, *) fl                        !  T  T  T  T  T  F  F  F  F  F
```

Пример 3. Присваивание значений двумерному массиву.

```
integer  a(5, 2), i, j                ! Элементы списка в конструкторе массива b -
integer  b(3, 4)                      ! вложенные неявные циклы
a = reshape(source = (/ (2*i, i = 2, 11) /), shape = (/ 5, 2 /))
b = reshape((/ ((i*j, i = 1,3), j = 3, 6) /), shape = (/ 3, 4 /))
write(*, '(10i3)') a
write(*, '(4i3)') ((b(i, j), j = 1, 4), i = 1, 3)
```

Результат:

```
 4   6   8  10  12  14  16  18  20  22
 3   4   5   6
 6   8  10  12
 9  12  15  18
```

Пример 4. Смесь неявного списка и простых значений.

```
integer a(10), i
a = (/ 4, 7, (2*i, i = 1, 8) /)
write(*, '(10i3)') a                  !   4   7   2   4   6   8  10  12  14  16
```

Помимо использования в конструкторе массива функции RESHAPE, присваивание значений многомерному массиву можно также выполнить, последовательно применив несколько конструкторов массива, каждый раз определяя линейное сечение массива, например:

```
integer b(2, 3), i, j
b(1, :) = (/(i, i = 2, 6, 2)/)          ! Присвоим значения первому ряду
b(2, :) = (/5, 81, 17/)                 ! Присвоим значения второму ряду
write(*, '(3i3)') ((b(i, j), j = 1, 3), i = 1, 2)
```

Результат:

```
 2   4   6
 5  81  17
```

Помимо присваивания, массив можно изменить при выполнении операторов В/В (в массив можно вывести данные, поскольку он является внутренним файлом (разд. 10.3), а также при использовании массива в качестве фактического параметра процедуры.

4.7. Маскирование присваивания

4.7.1. Оператор и конструкция WHERE

В Фортране можно, используя оператор или конструкцию WHERE, выполнить присваивание только тем элементам массива, значения которых удовлетворяют некоторым условиям. Например:

```
integer :: b(5) = (/ 1, -1, 1, -1, 1 /)
where(b > 0) b = 2 * b
print *, b                              !    2    -1    2    -1    2
```

В Фортране 77 для подобных действий используется цикл

```
do k = 1, 5
 if(b(k) .gt. 0) b(k) = 2 * b(k)
end do
```

Синтаксис оператора WHERE:

WHERE(*логическое выражение - массив*) *присваивание массива*

Синтаксис конструкции WHERE:

WHERE(*логическое выражение - массив*)
  операторы присваивания массивов
END WHERE

WHERE(*логическое выражение - массив*)
  операторы присваивания массивов
ELSEWHERE
  операторы присваивания массивов
END WHERE

Первоначально вычисляется значение *логического выражения - массива*. Его результатом является логический массив, называемый *массивом-маской*, под управлением которого и выполняется выборочное *присваивание массивов*. Такое выборочное присваивание называется *маскированием присваивания*. Поскольку массив-маска формируется до

выполнения присваивания массивов, то никакие выполняемые в теле WHERE изменения над массивами, входящими в *логическое выражение - массив*, не передаются в массив-маску.

Присутствующие в WHERE массивы должны иметь одинаковую форму. Попытка выполнить в теле оператора или конструкции WHERE присваивание скаляра или массивов разной формы приведет к ошибке компиляции.

Значения присваиваются тем следующим после WHERE элементам массивов, для которых соответствующий им элемент массива маски равен .TRUE. Если значение элемента массива-маски равно .FALSE. и если в конструкции WHERE присутствует ELSEWHERE, то будет выполнено присваивание следующих после ELSEWHERE элементов массивов, соответствующих по порядку следования элементу массива-маски.

Пример:

```
integer :: a(10) = (/1, -1, 1, -1, 1, -1, 1, -1, 1, -1/)
integer :: b(-2:7) = 0
where(a > b)                    ! Массивы a и b одной формы
 b = b + 2
elsewhere
 b = b - 3
 a = a + b
end where
print '(10i3)', a               ! 1 -4  1 -4  1 -4  1 -4  1 -4
print '(10i3)', b               ! 2 -3  2 -3  2 -3  2 -3  2 -3
end
```

Присутствующие в теле WHERE элементные функции вычисляются под управлением массива-маски, т. е. в момент выполнения присваивания. Например,

```
real :: a(5) = (/1.0, -1.0, 1.0, -1.0, 1/)
where(a > 0) a = log(a)
```

не приведет к ошибке, поскольку встроенная элементная функция вычисления натурального логарифма будет вызываться только для положительных элементов массива. Следующий фрагмент также синтаксически правилен:

```
real :: a(5) = (/1.0, -1.0, 1.0, -1.0, 1.0/)
where(a > 0) a = a / sum(log(a))
```

но приведет к ошибке выполнения, поскольку маскирование не распространяется на неэлементные функции и функция SUM вычисления суммы элементов массива будет выполнена до выполнения оператора WHERE. Иными словами, приведенный фрагмент аналогичен следующему:

```
real :: a(5) = (/1.0, -1.0, 1.0, -1.0, 1.0/), s
integer k
```

```
s = sum(log(a))                  ! При вычислении суммы возникнет ошибка из-за
do k = 1, 5                      ! попытки найти логарифм отрицательного числа
  if(a(k) > 0) a(k) = a(k) / s
end do
```

Нельзя передавать управление в тело конструкции WHERE, например, посредством оператора GOTO.

Фортран 95 расширил возможности конструкции WHERE. Теперь она может включать оператор ELSEWHERE(*логическое выражение - массив*).

Пример. В векторе *a* к положительным элементам прибавить число 2, к отрицательным - число 3, а к равным нулю - число 4.

```
integer :: a(9) = (/1, 2, 3, -1, -2, -3, 0, 0, 0/)
where(a > 0)
 a = a + 2
elsewhere(a < 0)                 ! Эта возможность добавлена стандартом 1995 г.
 a = a + 3
elsewhere
 a = a + 4
end where
print '(10i3)', a                 ! 3 4 5 2 1 0 4 4 4
end
```

Кроме того, в CVF конструкция WHERE может иметь имя, употребляемое по тем же правилам, что и имя в конструкции DO или IF. Эта возможность CVF является расширением над стандартом.

4.7.2. Оператор и конструкция FORALL

Оператор и конструкция FORALL, наряду с сечениями массивов и оператором и конструкцией WHERE, используются для выборочного присваивания массивов. FORALL может заменить любое присваивание сечений или WHERE. Но возможности FORALL шире: оператором и особенно конструкцией FORALL можно выполнять присваивания несогласованных массивов, т. е. массивов разной формы. Подобно WHERE и сечениям FORALL заменяет циклы с присваиванием массивов, например вместо цикла

```
do i = 1, 100
  d(i, i) = 2 * g(i)
end do
```

лучше использовать

```
forall(i = 1:100) d(i, i) = 2 * g(i)
```

Синтаксис оператора:

FORALL(*спецификация триплета* &
[, *спецификация триплета*] ... &
[, *выражение-маска*]) *оператор присваивания*

Синтаксис конструкции:

[*имя*:] FORALL(*спецификация триплета* &
[, *спецификация триплета*] ... &
[, *выражение-маска*])
операторы конструкции FORALL
END FORALL [*имя*]

спецификация триплета имеет вид:

индекс = триплет

где триплет - это тройка: [*нижняя граница*]:[*верхняя граница*]:[*шаг*].

Каждый из параметров триплета является целочисленным выражением. Параметр *шаг* изменения индексов может быть и положительным и отрицательным, но не может быть равным нулю; *шаг*, если он отсутствует, принимается равным единице. Все параметры триплета являются необязательными. В выражениях, задающих нижнюю, верхнюю границы триплета и его шаг, не должно быть ссылок на индекс. Оценка какого-либо выражения триплета не должна влиять на результат его иного выражения.

индекс - это скаляр целого типа. Область видимости индекса - оператор или конструкция FORALL. После завершения FORALL значение индекса не определено.

выражение-маска - логическое выражение - массив; при отсутствии принимается равным .TRUE.. Содержит, как правило, имена индексов, например:

```
forall(i = 1:n, i = 1:n, a(i, j) /= 0.0) b(i, j) = 1.0 / a(i, j)
```

Переменная, которой в *операторе присваивания* присваиваются значения, должна быть элементом массива или его сечением и должна содержать имена всех индексов, включенных в спецификации триплетов. Правая часть оператора присваивания не может быть символьного типа.

имя - имя конструкции FORALL.

операторы конструкции FORALL - это:

- оператор присваивания, обладающий рассмотренными выше свойствами;
- оператор или конструкция WHERE;
- оператор или конструкция FORALL.

Присутствующие в FORALL операторы выполняются для тех значений индексов, задаваемых индексными триплетами, при которых *выражение-маска* вычисляется со значением .TRUE..

В DO-цикле операторы выполняются немедленно при каждой итерации. FORALL работает иначе: первоначально вычисляется правая часть выражения для всех итераций и лишь затем выполняется присваивание. То же справедливо и для выражений с сечениями, например:

```
integer(4), parameter :: n = 5
integer(4), dimension(n) :: a = 1          ! Объявляем и инициализируем массив a
```

```
integer(4) :: k
do k = 2, n                              ! Выполним присваивание в цикле
  a(k) = a(k - 1) + 2
end do
print *, a                               !    1    3    5    7    9
a = 1                                    ! Присваивание в FORALL
forall(k = 2:n) a(k) = a(k - 1) + 2
print *, a                               !    1    3    3    3    3
a = 1                                    ! Используем выражение с сечениями
a(2:n) = a(1:n-1) + 2
print *, a                               !    1    3    3    3    3
```

Ни один из элементов массива не может быть изменен в FORALL более одного раза.

Любая процедура, вызываемая в выражении-маске FORALL, должна быть чистой.

Любую конструкцию или оператор WHERE можно заменить FORALL, обратное утверждение несправедливо. Примером служит оператор

```
forall(i = 1:n, j = 1:n) h(i, j) = 1.0 / real(i + j)
```

в котором элементами выражения являются изменяемые индексы, что для WHERE недопустимо.

Пример 1:

```
type monarch
  integer(4), pointer :: p
end type monarch
type(monarch), dimension(8) :: pattern
integer(4), dimension(8), target :: object
forall(j = 1:8) pattern(j)%p => object(1 + ieor(j - 1, 2))
```

Этот оператор FORALL прикрепляет элементы с номерами 1-8 ссылки *pattern* соответственно к элементам 3, 4, 1, 2, 7, 8, 5 и 6 адресата *object*. Встроенная функция IEOR может быть использована, так как она, как и все встроенные процедуры, является чистой.

Пример 2. Использование именованной конструкции FORALL.

```
ex2: forall(i = 3:n + 1, j = 3:n + 1)
  c(i, j) = c(i, j + 2) + c(i, j - 2) + c(i + 2, j) + c(i - 2, j)
  d(i, j) = c(i, j)                      ! Массиву d присваиваются вычисленные
end forall ex2                           ! в предыдущем операторе элементы массива c
```

Пример 3. Операторы FORALL, которые не заменяются сечениями или WHERE.

```
real(4), dimension(100, 100) :: a = 1.0, b = 2.0
real(4), dimension(300) :: c = 3.0
integer(4) :: i, j
forall(i = 1:100, j = 1:100) a(i, j) = (i + j) * b(i, j)
forall(i = 1:100) a(i, i) = c(i)
```

Заметьте, что в последнем случае FORALL обеспечивает доступ к диагонали матрицы, чего нельзя сделать при помощи сечений массивов.

Пример 4. Формируется вектор *a* из сумм вида $\sum_{i=1}^{m} x_i$, $(m = 1, 2, \ldots, n)$.

```
program vec_a
 integer(4), parameter :: n = 10
 integer(4) i
 ! Инициализация массива x:
 real(4) :: x(n) = (/ (i, i = 1, n) /), a(n)
 ! Массив x после инициализации:
 ! 1.0  2.0  3.0  4.0  5.0  6.0  7.0  8.0  9.0  10.0
 forall(i = 1:n) a(i) = sum(x(1:i))
 print '(1x, 10f5.1)', a(1:n)          ! 1.0  3.0  6.0 10.0 15.0 21.0 28.0 36.0 45.0 55.0
end program vec_a
```

Замечание. Оператор и конструкция FORALL введены стандартом Фортран 95.

4.8. Динамические массивы

4.8.1. Атрибуты POINTER и ALLOCATABLE

При использовании статических массивов перед программистом всегда стоит проблема задания их размеров. В некоторых случаях эта проблема не имеет удовлетворительного решения. Так, если в массиве хранятся позиционные обозначения элементов печатной платы или интегральной схемы, то в зависимости от сложности проектируемого устройства может потребоваться массив размером от 10 до 1000 и более элементов. В таком случае лучше использовать динамические массивы, размеры которых можно задать и изменить в процессе выполнения программы. В Фортране существует 3 вида динамических массивов: массивы-ссылки, размещаемые массивы и автоматические массивы.

Динамические массивы-ссылки объявляются с атрибутом POINTER, который может быть задан или в операторе объявления типа, или посредство оператора POINTER. Размещаемые массивы объявляются с атрибутом ALLOCATABLE, который задается или в операторе объявления типа, или посредством оператора ALLOCATABLE.

Автоматические массивы создаются в процедурах, и их размер определяется при вызове процедуры.

Память под массивы-ссылки и размещаемые массивы может быть выделена оператором ALLOCATE. Напомним, что выделяемая под массив-ссылку оператором ALLOCATE память называется адресатом ссылки. Массив-ссылка также получает память после его прикрепления к уже имеющему память адресату (разд. 3.11.2). Например:

```
real, pointer :: a(:), c(:), d(:)       ! Одномерные массивы-ссылки
integer, allocatable :: b(:, :)         ! Двумерный размещаемый массив
real, allocatable, target :: t(:)       ! Размещаемый массив-адресат
real a3
allocatable a3(:, :, :)                 ! Оператор ALLOCATABLE
real, target :: t2(20)                  ! Статический массив-адресат
allocate(a(10), a2(10*2), a3(2, 2, 5), b(5, 10), t(-30:30))
c => t                                  ! Прикрепление ссылок к динамическим
d => c                                  ! ранее получившим память адресатам
c => t2                                 ! Прикрепление ссылки к статическому адресату
```

4.8.2. Операторы ALLOCATE и DEALLOCATE

Оператор ALLOCATE создает адресаты ссылок и размещает массивы, заданные с атрибутом ALLOCATABLE. Синтаксис оператора:

ALLOCATE(*var* | *array*(*shape_spec_list*) &
[, *var* | *array*(*shape_spec_list*)).[, STAT = *ierr*])

var - имя размещаемой ссылки, которое может быть компонентом структуры.

array - имя массива-ссылки или размещаемого массива, которое может быть компонентом структуры.

shape_spec_list - список вида ([*lo*:]*up*[, [*lo*:]*up*] ...]), задающий форму размещаемого массива. Число задающих границы размерности пар должно совпадать с рангом размещаемого массива или массива-ссылки. Параметры *lo* и *up* являются целыми скалярными выражениями и определяют соответственно нижнюю и верхнюю границу протяженности по соответствующему измерению массива. Если параметр *lo*: опущен, то по умолчанию нижняя граница принимается равной единице. Если *up* < *lo*, размер массива равен нулю.

STAT - параметр, позволяющий проконтролировать, удалось ли разместить массив. Любая неудача, если не задан параметр STAT, приводит к ошибке этапа исполнения и остановке программы. Параметр указывается в операторе ALLOCATE последним.

ierr - целочисленная статусная переменная, возвращающая 0, если размещение выполнено удачно; в противном случае возвращается код ошибки размещения, например:

```
integer, allocatable :: b(:, :)
integer m/5/, n/10/, ierr
allocate(b(m, n), stat = ierr)
if(ierr .ne. 0) then
 write(*, *) 'Ошибка размещения массива b. Код ошибки: ', ierr
 stop
end if
```

Причиной ошибки может быть, например, недостаток памяти или попытка разместить ранее размещенный и не освобожденный оператором DEALLOCATE объект.

Оператор ALLOCATE выделяет память и задает форму массиву или адресату ссылки. Для освобождения выделенной под размещаемый массив памяти используется оператор DEALLOCATE. После освобождения памяти размещаемый массив может быть размещен повторно с новым или прежним размером.

Поскольку порядок выделения памяти не регламентируется, то спецификация формы массива не может включать справочную функцию массива, имеющую аргументом массив того же оператора ALLOCATE. Так, ошибочен оператор

```
allocate(a(10), b(size(a)))          ! Неверно
```

Вместо него следует задать два оператора:

```
allocate(a(10))
allocate(b(size(a)))                 ! Правильно
```

Если присутствует параметр STAT= и статусная переменная *ierr* является динамической, то ее размещение должно быть выполнено в другом, предшествующем операторе ALLOCATE.

Изменение переменных, использованных для задания границ массива, после выполнения оператора ALLOCATE не окажет влияния на заданные границы. Например:

```
n = 10
allocate(a(n:2*n))                   ! Размещение массива a(10:20)
n = 15                               ! Форма массива a не меняется
```

Размещаемый массив может иметь статусы "не размещен", "размещен" и "не определен". Попытка адресовать массив со статусом "не размещен" приводит к непредсказуемым результатам. Для определения, размещен ли размещаемый массив, используется встроенная функция ALLOCATED, возвращающая значение стандартного логического типа, равное .TRUE., если массив размещен, и .FALSE. - в противном случае. (Напомним, что для определения статуса ссылки используется функция ASSOCIATED.) Например:

```
real, allocatable :: b(:, :)
real, pointer :: c(:, :)
integer :: m = 5, n = 10
allocate(b(m, n), c(m, n))
if(allocated(b)) b = 2.0
if(associated(c)) c = 3.0
```

При использовании размещаемого массива в качестве локальной переменной процедуры следует перед выходом из процедуры выполнить

освобождение отведенной для него памяти (в противном случае размещаемый массив приобретет статус "не определен"):

```
subroutine delo(n)
 integer n, ierr
 integer, allocatable :: ade(:)
 allocate(ade(n), stat = ierr)
 if(ierr /= 0) stop 'Allocation error'
 ...
 deallocate(ade)
end subroutine delo
```

Объявленный в модуле размещаемый массив после размещения в использующей модуль программной единице имеет статус "размещен" в любой другой использующей (после размещения массива) этот модуль программной единице. Например:

```
module wara
 integer n
 integer, allocatable :: work(:, :)
end module wara

program two
 use wara
 call rewo( )                    ! Выделяем память под массив work
 work = 2                        ! Массив work размещен в подпрограмме rewo
 print *, work(5, 5)             !       2
 call delo(4)
 print *, work(5, 5)             !       4
end program two

subroutine rewo( )
 use wara
 print  '(1x, a\)',  'Enter n: '
 read(*, *) n
 allocate(work(n, 2 * n))
end subroutine rewo

subroutine delo(k)
 use wara
 integer k
 work(5, 5) = k                  ! Размещать work не нужно, поскольку
end subroutine delo              ! он размещен в подпрограмме rewo
```

Таким же свойством обладают и ссылки, т. е. если в последнем примере в модуле *wara* объявить ссылку

```
integer, pointer :: work(:, :)
```

то выделенный ей в подпрограмме *rewo* адресат будет доступен и в главной программе, и в подпрограмме *delo*.

Попытка разместить уже размещенный массив всегда приводит к ошибке. Однако прикрепленную (ранее размещенную) ссылку можно разместить повторно, в результате чего она прикрепляется к новому, созданному оператором ALLOCATE адресату, например:

```
real, pointer :: a(:)
allocate(a(10))          ! Размещение ссылки a
allocate(a(20))          ! Прикрепление ссылки a к другой области памяти
```

Однако такое переразмещение ссылки приведет к созданию неиспользуемой и недоступной памяти. Чтобы этого избежать, следует до переназначения ссылки выполнить оператор

```
deallocate(a)
```

Оператор DEALLOCATE освобождает выделенную оператором ALLOCATE память и имеет синтаксис

DEALLOCATE(*a-list* [, STAT = *ierr*])

a-list - список из одного или более имен размещенных объектов (массивов, массивов-ссылок или простых ссылок), разделенных запятыми. Все элементы списка должны быть ранее размещены оператором ALLOCATE либо, в случае ссылок, должны иметь созданного оператором ALLOCATE адресата. Адресатом в списке *a-list* ссылки должен являться полный объект (например, адресатом освобождаемой DEALLOCATE ссылки не может быть сечение, элемент массива, подстрока). Напомним, что откреплeние ссылки, получившей не созданного оператором ALLOCATE адресата, выполняется оператором NULLIFY.

STAT - необязательный параметр, позволяющий проконтролировать, удалось ли освободить память. Любая неудача, если не задан параметр STAT, приводит к ошибке исполнения и к остановке программы. Параметр STAT должен появляться в операторе последним.

ierr - целочисленная переменная, возвращающая 0, если память удалось освободить; в противном случае возвращается код ошибки. Если *ierr* является динамической переменной, то она не может быть освобождена использующим ее оператором DEALLOCATE.

Попытка освобождения неразмещенного объекта вызовет ошибку выполнения.

Если оператором DEALLOCATE выполнено освобождение массива с атрибутом TARGET, то статус подсоединенной к массиву ссылки становится неопределенным и к ней можно обращаться только после ее прикрепления к другому объекту.

Пример:

```
integer, allocatable :: ade(:)
real, pointer :: ra, b(:)
allocate(ade(10), ra, b(20))      ! Размещаем массивы ade и b
```

```
...                              ! и ссылку-скаляр ra
deallocate(ade, ra, b)           ! Освобождаем динамическую память
```

Замечания:

1. Хорошей практикой является освобождение оператором DEALLOCATE выделенной динамической памяти, когда надобность в ней отпадает. Это позволяет избежать накопления неиспользуемой и недоступной памяти.

2. Фортран 95 ввел автоматическое освобождение размещаемых массивов, объявленных в процедурах. Оно работает так: если при выходе из процедуры пользователь явно оператором DEALLOCATE не освободил занимаемую размещаемым массивом память, то теперь это произойдет автоматически. Это новое свойство делает язык более надежным. Например, в FPS, который не поддерживает стандарт Фортран 95, следующий код

```
program t4
 call a( )
 call a( )                       ! Второй вызов в FPS завершится ошибкой
end program t4

subroutine a( )
  real, allocatable :: b (:)
  print *, allocated(b)          ! FPS при первом вызове напечатает F, при втором - T
  allocate(b(10))                ! CVF в обоих вызовах напечатает F
end
```

приведет к возникновению ошибки: run-time error F6316 - array already allocated, которая приведет к прекращению вычислений. В CVF оба вызова завершатся благополучно.

4.8.3. Автоматические массивы

В процедуре может быть задан локальный массив, размеры которого меняются при разных вызовах процедуры. Такие массивы, так же как и локальные строки переменной длины (разд. 3.8.3), относятся к автоматическим объектам.

Пример. Создать процедуру обмена содержимого двух массивов.

```
program shos
integer, parameter :: n = 5
integer k
real :: a(n) = (/ (1.0, k = 1, n) /), b(n) = (/ (2.0, k = 1, n) /)
interface
 subroutine swap(a, b)           ! При использовании перенимающих форму
  real a(:), b(:)                ! массивов требуется задание явного интерфейса
 end subroutine swap
end interface
call swap(a, b)
```

```
write(*, *) b
end
subroutine swap(a, b)
 real a(:), b(:)                ! a и b - массивы, перенимающие форму
 real c(size(a))                ! c - автоматический массив
 c = a
 a = b
 b = c
end subroutine swap
```

Замечание. Если оформить *swap* как внутреннюю подпрограмму программы *shos*, то не потребуется задавать интерфейсный блок к *swap*, поскольку внутренние (равно как и модульные) процедуры обладают явно заданным интерфейсом.

К автоматическим объектам относятся объекты данных, размеры которых зависят от неконстантных описательных выражений (разд. 5.6) и которые не являются формальными параметрами процедуры. Такие объекты не могут иметь атрибуты SAVE и STATIC.

Границы автоматического массива или текстовая длина автоматической строки фиксируются на время выполнения процедуры и не меняются при изменении значения соответствующего выражения описания.

4.9. Массивы - формальные параметры процедур

В процедурах форма и размер массива - формального параметра могут определяться в момент вызова процедуры. Можно выделить 3 вида массивов - формальных параметров: заданной формы, перенимающие форму и перенимающие размер.

4.9.1. Массивы заданной формы

Границы размерностей массивов - формальных параметров могут определяться передаваемыми в процедуру значениями других параметров. Например:

```
integer, parameter :: n = 5, m = 10, k = m * n
real a(m, n) / k*1.0 /, b(m, n) / k*2.0 /
call swap(a, b, m, n)
write(*, *) b
end
subroutine swap(a, b, m, n)
 integer m, n                   ! a и b - массивы заданной формы
 real a(m*n), b(m*n)
 real c(size(a))                ! c - автоматический массив
 c = a
 a = b
```

```
 b = c
end subroutine swap
```

Такие массивы - формальные параметры называются *массивами заданной формы*. В примере их форма задается формальными параметрами *m* и *n*.

Из примера видно, что формы фактического и соответствующего ему формального параметра - массива могут отличаться. В общем случае могут вычисляться как нижняя, так и верхняя граница размерности. Общий вид размерности таких массивов:

[*нижняя граница*] : [*верхняя граница*]

Нижняя и верхняя границы - целочисленные описательные выражения (разд. 5.6).

Вычисленные границы массива фиксируются на время выполнения процедуры и не меняются при изменении значения соответствующего описательного выражения.

При работе с такими массивами необходимо следить, чтобы размер массива - формального параметра не превосходил размера ассоциированного с ним массива - фактического параметра.

Если фактическим параметром является многомерный массив и соответствующим ему формальным параметром является массив заданной формы с тем же числом измерений, то для правильного ассоциирования необходимо указать размерности массива - формального параметра такими же, как и у массива - фактического параметра. Исключение может составлять верхняя граница *последней* размерности массива, которая может быть меньше соответствующей границы массива - фактического параметра.

Если в качестве фактического параметра задан элемент массива, то формальный параметр ассоциируется с элементами массива-родителя начиная с данного элемента, и далее по порядку.

Пример. Вывести первый отрицательный элемент каждого столбца матрицы.

```
ineger, parameter :: m = 4, n = 5
integer j
real :: a(m, n) = 1.0
a(1, 1) = -1.0; a(2, 2) = -2.0; a(3, 3) = -3.0
do j = 1, n
 call prifin(a(1, j), m , j)          ! В prifin доступны все элементы
end do                                ! столбца j начиная с первого и все
end                                   ! последующие элементы матрицы a

subroutine prifin(b, m, j)
 integer m, i, j
 real b(m)                            ! Вектор b содержит все элементы
 do i = 1, m                          ! столбца j матрицы a
  if(b(i) < 0) then
```

```
  print *, 'Столбец ', j, '.  Элемент ', b(i)
  return
 end if
end do
print *, 'В столбце ', j, ' нет отрицательных элементов'
end subroutine prifin
```

4.9.2. Массивы, перенимающие форму

Такие массивы - формальные параметры перенимают форму у соответствующего фактического параметра. В результате форма фактического и формального параметров совпадают. (Понятно, что фактический и формальный параметры должны быть одного ранга.) При описании формы формального параметра каждая размерность имеет вид:

[*нижняя граница*] :

где нижняя граница - это целое описательное выражение, которое может зависеть от данных в процедуре или других параметров. Если нижняя граница опущена, то ее значение по умолчанию равно единице. Например, при вызове

```
real x(0:3, 0:6, 0:8)
interface
 subroutine asub(a)
  real a(:, :, :)
 end
end interface
...
call asub(x)
```

соответствующий перенимающий форму массив объявляется так:

```
subroutine asub(a)
 real a(:, :, :)
 print *, lbound(a, 3), ubound(a, 3)              !      1       9
```

Так как нижняя граница в описании массива *a* отсутствует, то после вызова подпрограммы в ней будет определен массив *a*(4, 7, 9). Если нужно сохранить соответствие границ, то массив *a* следует объявить так:

```
real a(0:, 0:, 0:)
```

В интерфейсном блоке по-прежнему массив *a* можно объявить:

```
real a(:, :, :)
```

Процедуры, содержащие в качестве формальных параметров перенимающие форму массивы, должны обладать явно заданным интерфейсом.

Если для работы с массивом в процедуре нужно знать значения его границ, то их можно получить, использовав функции LBOUND и UBOUND (разд. 4.12.3.2).

Пример. Даны матрицы *a* и *b* разной формы. В какой из них первый отрицательный элемент последнего столбца имеет наибольшее значение?

```
program neg2
interface                       ! Процедура, формальными параметрами которой
 integer function FindNeg(c)    ! являются перенимающие форму массивы,
  integer :: c(:, :)            ! должна обладать явным интерфейсом
 end function FindNeg
end interface
integer a(4, 5), b(-1:7, -1:8)
integer nga, ngb, ngmax         ! nga и ngb - соответственно последние
a = 1; a(3, 5) = -2             ! отрицательные элементы матриц a и b
b = 2; b(2, 8) = -3
nga = FindNeg(a)                ! Ищем nga и ngb
ngb = FindNeg(b)
if(nga == 0) print *, 'В последнем столбце матрицы a нет отрицательных элементов'
if(ngb == 0) print *, 'В последнем столбце матрицы b нет отрицательных элементов'
ngmax = max(nga, ngb)
! Если не все матрицы имеют отрицательный элемент в последнем столбце - STOP
if(ngmax == 0) stop
if(nga == ngmax) print *, 'В матрице a'
if(ngb == ngmax) print *, 'В матрице b'
end

integer function FindNeg(c)     ! Возвращает первый отрицательный элемент
 integer :: c(:, :)             ! последнего столбца матрицы c
 integer cols, i
 FindNeg = 0
 cols = ubound(c, dim = 2)      ! Номер последнего столбца в массиве c
 do i = lbound(c, 1), ubound(c, 1)
  if(c(i, cols) < 0) then
   FindNeg = c(i, cols)         ! Вернем найденный отрицательный элемент
   return
  end if
 end do
end function FindNeg
```

Результат:

```
В матрице a
```

4.9.3. Массивы, перенимающие размер

В перенимающем размер массиве - формальном параметре вместо верхней границы последней размерности проставляется звездочка (*). (Таким же образом задаются перенимающие длину строки.) Остальные границы должны быть описаны явно. Перенимающий размер массив может отличаться по рангу и форме от соответствующего массива - фактического параметра. Фактический параметр определяет только размер массива - формального параметра. Например:

```
real x(3, 6, 5), y(4, 10, 5)
call asub(x, y)
...
subroutine asub(a, b)
 real a(3, 6, *), b(0:*)
 print *, size(a, 2)                    !     6
 print *, lbound(a, 3), lbound(b)       !     1     0
```

Перенимающие размер массивы не имеют определенной формы. Это можно проиллюстрировать примером:

```
real x(7)
call assume(x)
...
subroutine assume(a)
 real a(2, 2, *)
```

При такой организации данных между массивами *x* и *a* устанавливается соответствие:

```
x(1) = a(1, 1, 1)
x(2) = a(2, 1, 1)
x(3) = a(1, 2, 1)
x(4) = a(2, 2, 1)
x(5) = a(1, 1, 2)
x(6) = a(2, 1, 2)
x(7) = a(1, 2, 2)
```

т. е. в массиве *a* нет элемента *a*(2, 2, 2). (Размер массива *a* определяется размером массива *x* и равен семи.)

Так как перенимающие размер массивы не имеют формы, то нельзя получить доступ ко всему массиву, передавая его имя в процедуру. (Исключение составляют процедуры, не требующие форму массива, например встроенная функция LBOUND.) Так, нельзя использовать только имя перенимающего размер массива в качестве параметра встроенной функции SIZE, но можно определить протяженность вдоль фиксированной (т. е. любой, кроме последней) размерности.

Можно задать сечения у перенимающего размер массива. Однако в общем случае надо следить, чтобы все элементы сечения принадлежали массиву. Так, для массива *a* из последнего примера нельзя задать сечение *a*(2, 2, 1:2), поскольку в массиве *a* нет элемента *a*(2, 2, 2).

Необходимо следить, чтобы при работе с перенимающими размер массивами не происходило обращений к не принадлежащим массиву ячейкам памяти. Так, следующий фрагмент будет выполнен компьютером, но исказит значение переменной *d*:

```
program bod
 integer b(5) /5*11/, d /-2/
 call testb(b)
```

```
 write(*, *) d                  ! Вместо ожидаемого -2 имеем 15
end
subroutine testb(b)
 integer b(*)                   ! Массив, перенимающий размер
 b(6) = 15                      ! По адресу b(6) находится переменная d
end
```

Учитывая отмеченные недостатки в организации перенимающих размер массивов (отсутствие формы, возможность выхода за границы, невозможность использования в качестве результата функции), следует применять в качестве формальных параметров массивы заданной формы или массивы, перенимающие форму.

4.10. Использование массивов

Фортран позволяет работать с массивами и сечениями массивов так же, как и с единичными объектами данных:

- массивам и их сечениям можно присваивать значения;
- массивы (сечения массивов) можно применять в качестве операндов в выражениях с арифметическими, логическими операциями и операциями отношения. Результат выражения, которое содержит массивы и/или сечения массивов, присваивается массиву (сечению). В результате присваивания получается массив (сечение), каждый элемент которого имеет значение, равное результату операций над соответствующими элементами операндов:

  массив | *сечение* = *выражение*
- массивы и сечения могут быть параметрами встроенных элементных функций (разд. 6.1), например SIN и SQRT. Элементные функции, получив массив (сечение) в качестве аргумента, выполняются последовательно для всех элементов массива (сечения).

Правда, существует ограничение: используемые в качестве операндов массивы (сечения массивов) и массив (сечение), которому присваивается результат выражения, должны быть согласованы.

Массивы (сечения) *согласованы*, если они имеют одинаковую форму. Так, согласованы массивы *a*(-1:6) и *b*(-1:6), массивы *c*(2:9) и *b*(-1:6). Всегда согласованы массив и скаляр.

Пример:

```
integer(2) a(5) /1, 2, 3, 4,  5/, a2(4, 7) /28*5/
integer(2) :: b(5) = (/ 1, 2, -2, 4, 3 /)
integer(2) :: ic(5) = 1
character(1) d1(4) / 'a', 'c', 'e', 'g' /
character(1) d2(4) / 'b', 'd', 'f', 'h' /
character(4) d12(4)
real(4) c(2) /2.2, 2.2/, d(2) /2.2, 3.3/, cd*8(2)
```

```
logical(1) g(3) / .true., .false., .true. /
logical(1) h(3) / 3*.false. /
ic = ic + 2 * (a - b) - 2          ! Присваивание массива
write(*, *) ic                     !     -1      -1       9      -1       3
a2(3, 1:5) = a2(3, 1:5) - a        ! Согласованные массив и сечение
write(*, *) int(sqrt(real(a*b**2))) !      1       2       3       8       6
write(*, *) exp(real(a/b))         !  2.718   2.718  3.68e-01     ...
write(*, *) a**b                   !      1       4       0     256     125
d12 = d1 // '5' // d2 // '5'
write(*, *) d12                    ! a5b5c5d5e5f5g5h5
write(*, *) mod(a, b)              !      0       0       1       0       2
write(*, *) sign(a, b)             !      1       2      -3       4       5
write(*, *) dim(a, b)              !      0       0       5       0       2
cd = dprod(c, d)                   !  4.840     ...    7.26
write(*, *) g .and. .not. h        !      T       F       T
end
```

4.11. Массив как результат функции

Массив также может быть и результатом функции. Интерфейс к такой функции должен быть задан явно. Если функция не является ссылкой, то границы массива должны быть описательными выражениями, которые вычисляются при входе в функцию.

Пример. Составить функцию, возвращающую массив из первых *n* положительных элементов передаваемого в нее массива.

```
real, allocatable :: ap(:)
real :: a(10) = (/ 1, -1, 2, -2, 3, -3, 4, -4, 5, -5 /)
real :: b(10) = (/ 2, -1, 3, -2, 4, -3, 4, -4, 5, -5 /)
integer n /3/
allocate(ap(n))
ap = fap(a, n) + fap(b, n)         ! Вызов массивоподобной функции fap
print '(1x, 10f5.1)', ap           !  3.0  5.0  7.0
contains
 function fap(a, n)
  integer :: n, k, i
  real :: fap(n)                   ! Результатом функции является массив
  real, dimension(:) :: a          ! a - перенимающий форму массив
  fap = 0
  k = 0
  do i = 1, size(a)
   if(a(i) > 0) then
    k = k + 1
    fap(k) = a(i)                  ! Формирование массива-результата
    if(k == n) exit
   end if
  end do
```

```
 end function fap
end
```

Замечания:

1. Если оформить функцию *fap* как внешнюю, то к ней нужно в вызывающей программной единице явно задать интерфейс:

```
interface
 function fap(a, n)
  integer n
  real fap(n)
  real, dimension(:) :: a
 end function fap
end interface
```

2. Ту же задачу можно решить, не создавая довольно громоздкой функции *fap*, а дважды вызвав встроенную функцию PACK (разд. 4.12.4.2):

```
ap = pack(a, a > 0) + pack(b, b > 0)
```

Однако если длина возвращаемых функцией PACK массивов меньше *n*, часть элементов массива *ap* будет не определена, а результат - непредсказуем.

Функции, возвращающие массив, называют *массивоподобными функциями*. Из примера видно, что массивоподобные функции могут быть, как и обычные функции, операндами выражений.

4.12. Встроенные функции для массивов

В Фортран встроено большое число функций, позволяющих:

- выполнять вычисления в массивах, например находить максимальный элемент массива или суммировать его элементы;
- преобразовывать массивы, например можно получить из одномерного массива двумерный;
- получать справочные данные о массиве (размер, форма и значения границ каждого измерения).

Вызов любой функции для массивов может быть выполнен с ключевыми словами, в качестве которых используются имена формальных параметров. Вызов с ключевыми словами обязателен, если позиции соответствующих фактического и формального параметров не совпадают.

Параметрами всех рассматриваемых в разделе функций могут быть и сечения массивов.

Некоторые функции возвращают результат стандартного целого или логического типа. По умолчанию значение параметра разновидности для этих типов KIND равно четырем. Однако если задана опция компилятора /4I2 или директива $INTEGER:2, значение разновидности стандартного целого и логического типов будет равно двум.

Как и ранее, необязательные параметры функций обрамляются квадратными скобками.

4.12.1. Вычисления в массиве

ALL(*mask* [, *dim*]) - возвращает .TRUE., если все элементы логического массива *mask* вдоль заданного (необязательного) измерения *dim* истинны; в противном случае возвращает .FALSE.

Результатом функции является логический скаляр, если *mask* одномерный массив или опущен параметр *dim* (в этом случае просматриваются все элементы массива *mask*). Иначе результатом является логический массив, размерность которого на единицу меньше размерности *mask*. Разновидности типа результата и *mask* совпадают.

Параметр *dim* - целое константное выражение ($1 \le dim \le n$, где n - размерность массива *mask*). Параметр *dim*, если он задан, означает, что действие выполняется по всем одномерным сечениям, которые можно задать по измерению с номером *dim*. Функция вычисляет результат для каждого из сечений и заносит в массив на единицу меньшего ранга с экстентами, равными экстентам по остальным измерениям. Так, в двумерном массиве *mask*(1:2, 1:3) по второму измерению (*dim* = 2) можно задать два одномерных сечения: *mask*(1, 1:3) и *mask*(2, 1:3). Поэтому для хранения результата функции ALL(*mask*, 2) следует использовать одномерный логический массив из двух элементов.

ANY(*mask* [, *dim*]) - возвращает .TRUE., если хотя бы один элемент логического массива вдоль заданного (необязательного) измерения *dim* истинен; в противном случае функция возвращает .FALSE.

Результат функции и действие параметра *dim* определяются по тем же правилам, что и для функции ALL.

COUNT(*mask* [, *dim*]) - возвращает число элементов логического массива *mask*, имеющих значение .TRUE. вдоль заданного необязательного измерения *dim*. Результат функции имеет стандартный целый тип. Правила получения результата и действие параметра *dim* такие же, как и для функции ALL.

Для функций ALL, ANY, COUNT:

Пример 1:

```
logical ar1(3), ar2(2)              ! Массивы для хранения результатов
logical mask(2, 3) / .true., .true., .false., .true., .false., .false. /

 ! Массив mask:       .true.   .false.   .false.

 !                    .true.   .true.    .false.

ar1 = all(mask, dim = 1)            ! Оценка элементов в столбцах массива
print *, ar1                        ! T F F
ar2 = all(mask, dim = 2)            ! Оценка элементов в строках массива
print *, ar2                        ! F F
```

```
print *, any(mask, dim = 1)      ! T T F
print *, any(mask, dim = 2)      ! T T
print *, count(mask, dim = 1)    !    2    1    0
print *, count(mask, dim = 2)    !    1    2
end
```

Пример 2. Если хотя бы один элемент второго столбца матрицы *a* меньше нуля, напечатать число положительных элементов ее первой строки.

```
integer a(4, 5)
a = 2;                           ! Инициализация массива
a(2, 2) = 0                      ! Теперь второй элемент второго столбца равен нулю
a(1, 3) = -1                     ! Теперь третий элемент первой строки меньше нуля
if(any(mask = a(:, 2) == 0)) print *, count(a(1, :) > 0)
end                              !    4
```

MAXLOC(*array* [, *dim*] [, *mask*]) - возвращает индексы максимального элемента массива *array* или максимальных элементов по заданному измерению *dim*. Значение каждого максимального элемента удовлетворяет заданным (необязательным) условиям *mask*. Если несколько элементов содержат максимальное значение, то берется первый по порядку их следования в *array*. Результат MAXLOC, если не задан параметр *dim*, записывается в одномерный массив, размер которого равен числу измерений *array*.

Параметр *mask* является логическим массивом, форма которого совпадает с формой массива *array*. Массив *mask* может получаться в результате вычисления логического выражения. Если массив задан, то действие функции распространяется только на те элементы массива *array*, для которых значение *mask* равно .TRUE. Если же параметр *mask* опущен, то действие функции распространяется на все элементы массива *array*.

Значения индексов берутся так, словно все нижние границы массива *array* равны единице. Если же маска такова, что наибольший элемент не может быть найден, то возвращаемые значения индексов превышают верхнюю границу каждой размерности массива *array*.

Если параметр *dim* задан, то:

- массив-результат имеет ранг, на единицу меньший ранга массива *array*, и форму (d_1, d_2, ..., d_{dim-1}, d_{dim+1}, ..., d_n), где (d_1, d_2, ..., d_n) - форма массива *array*;
- если *array* имеет ранг, равный единице, то MAXLOC(*array*, *dim* [, *mask*]) возвращает то же, что и функция MAXLOC(*array* [, MASK = *mask*]); иначе значение элемента (s_1, s_2, ..., s_{dim-1}, s_{dim+1}, ..., s_n) результата функции MAXLOC(*array*, *dim* [, *mask*]) равно MAXLOC(*array*(s_1, s_2, ..., s_{dim-1}, s_{dim+1}, ..., s_n), [, MASK = *mask*(s_1, s_2, ..., s_{dim-1}, s_{dim+1}, ..., s_n)]).

MINLOC(*array* [, *dim*] [, *mask*]) выполняет те же действия, что и MAXLOC, но для минимальных элементов массива *array*. Смысл параметров *dim* и *mask* такой же, что и для функции MAXLOC. Значения индексов берутся так, словно все нижние границы массива *array* равны единице. Если же маска такова, что наименьший элемент не может быть найден, то возвращаемые значения индексов превышают верхнюю границу каждой размерности массива *array*.

Замечание. Необязательный параметр *dim* добавлен в функции MAXLOC и MINLOC стандартом 1995 г.

Пример для функций MAXLOC и MINLOC:

```
integer ir, maxf(1)
integer arra(3, 3) /7, 9, -1, -2, 5, 0, 3, 6, 9/
integer, allocatable :: ar1(:)
! Массив arra:      7    -2     3
!                   9     5     6
!                  -1     0     9
ir = size(shape(arra))              ! Ранг массива array (ir = 2)
allocate(ar1(ir))
! Найдем в массиве array индексы наибольшего, но меньшего семи элемента
ar1 = maxloc(arra, mask = arra < 7)
! Результатом выражения mask = arra < 7 является массив mask
! такой же формы, которую имеет и массив arra. Элементы массива mask имеют
! значение .TRUE., если соответствующий элемент массива arra меньше семи,
! и .FALSE. - в противном случае. Благодаря такой маске функция
! MAXLOC возвращает индексы максимального, но меньшего семи элемента
! Массив arra:      7    -2     3   Массив mask:     .false.   .true.   .true.
!                   9     5     6                    .false.   .true.   .true.
!                  -1     0     9                     .true.   .true.   .false.
print *, ar1                                  !    2     3
print *, minloc(arra, mask = arra > 0)        !    1     3
maxf = maxloc((/ 1, 4, 1, 4 /))
print *, maxf                                 !    2 (индекс первого максимума)
print *, minloc((/ 1, 4, 1, 4 /))             !    1 (индекс первого минимума)
end
```

Пример для функции MAXLOC с параметром *dim*:

```
integer(4), parameter :: m = 3, n = 5
real(4) :: a(m, n)
integer ip(n)
a = reshape((/    3.0,   4.0,   5.0,   6.0,   7.0, &
                  2.0,   3.0,   4.0,   5.0,   6.0, &
                  1.0,   2.0,   3.0,   4.0,    5.0 /), shape = (/ m, n /), order = (/ 2, 1 /))
```

```
ip = maxloc(array = a, mask = a < 5, dim = 1)
print *, ip                                  !    1    1    2    3    0
```

MAXVAL(*array* [, *dim*] [, *mask*]) - возвращает максимальное, удовлетворяющее необязательной маске *mask* значение целочисленного или вещественного массива *array* вдоль заданного необязательного измерения *dim*.

Смысл параметра *dim* разъяснен при описании функции ALL, а параметра *mask* - при описании функции MAXLOC.

Возвращаемое значение имеет тот же тип и разновидность типа, что и массив *array*. Если параметр *dim* опущен или массив *array* одномерный, то результатом функции MAXVAL является скаляр, в противном случае результатом является массив, ранг которого на единицу меньше ранга массива *array*.

Если размер массива 0 или все элементы массива *mask* равны .FALSE., то функция MAXVAL возвращает наибольшее по абсолютной величине отрицательное допускаемое процессором число.

MINVAL(*array* [, *dim*] [, *mask*]) - возвращает минимальное, удовлетворяющее необязательной маске *mask* значение целочисленного или вещественного массива *array* вдоль заданного необязательного измерения *dim*.

Смысл параметра *dim* разъяснен при описании функции ALL, а параметра *mask* - при описании функции MAXLOC.

Возвращаемое значение имеет тот же тип и разновидность типа, что и массив *array*. Если параметр *dim* опущен или массив *array* одномерный, то результатом функции MINVAL является скаляр, в противном случае результатом является массив, ранг которого на единицу меньше ранга массива *array*.

Если размер массива 0 или все элементы массива *mask* равны .FALSE., то функция MINVAL возвращает наибольшее положительное допускаемое процессором число.

Пример для функций MAXVAL и MINVAL:

```
integer array(2, 3), isha(2), max
integer, allocatable :: ar1(:), ar2(:)
array = reshape((/1, 4, 5, 2, 3, 6/), (/2, 3/))

! Массив array:    1  5  3
!                  4  2  6

isha = shape(array)                          ! isha = (2 3)
allocate(ar1(isha(2)))                       ! isha(2) = 3 - число столбцов в массиве
allocate(ar2(isha(1)))                       ! isha(1) = 2 - число строк в массиве
max = maxval(array, mask = array < 4)  ! Возвращает 3
ar1 = maxval(array, dim = 1)                 ! Возвращает (4  5  6)
ar2 = maxval(array, dim = 2)                 ! Возвращает (5  6)
print *, minval(array, mask = array > 3)  !    4     (ключевое слово mask обязательно)
```

```
print *, minval(array, dim = 1)          !   1   2   3
print *, minval(array, dim = 2)          !   1   2
! В следующем вызове все ключевые слова опущены
print *, minval(array, 1, array /= 2)    !   1   5   3
end
```

PRODUCT(*array* [, *dim*] [, *mask*]) - вычисляет произведение всех элементов целочисленного или вещественного массива вдоль необязательного измерения *dim*. Перемножаемые элементы могут отбираться необязательной маской *mask*.

Смысл параметра *dim* разъяснен при описании функции ALL, а параметра *mask* - при описании функции MAXLOC.

Возвращаемый функцией результат имеет тот же тип и разновидность типа, что и массив *array*.

Если размер массива *array* равен нулю или все элементы массива *mask* равны .FALSE., то результат функции равен единице.

SUM(*array* [, *dim*] [, *mask*]) - вычисляет сумму всех элементов целочисленного или вещественного массива вдоль необязательного измерения *dim*. Суммируемые элементы могут отбираться необязательной маской *mask*.

Смысл параметра *dim* разъяснен при описании функции ALL, а параметра *mask* - при описании функции MAXLOC.

Возвращаемый функцией результат имеет тот же тип и разновидность типа, что и массив *array*.

Если размер массива *array* равен нулю или все элементы массива *mask* равны .FALSE., то результат функции равен нулю.

Пример для функций PRODUCT и SUM:

```
integer arra (2, 3) /1, 4, 2, 5, 3, 6/
integer ar1(3), ar2(2)
! Массив array:  1  2  3
!                4  5  6
ar1 = product(arra, dim = 1)                  ! Произведение по столбцам
print *, ar1                                  !       4       10       18
ar2 = product(arra, mask = arra < 6, dim = 2)
print *, ar2                                  !       6       20
print *, sum(arra, dim = 1)                   !       5        7        9
ar2=sum(arra, mask = arra < 6, dim = 2)       ! Суммирование по строчкам
print *, ar2                                  !       6        9
! Произведение сумм столбцов матрицы: (1 + 4) * (2 + 5) * (3 + 6)
print *, product(sum(arra, dim = 1))          !    315
end
```

4.12.2. Умножение векторов и матриц

DOT_PRODUCT(*vector_a*, *vector_b*) - функция возвращает скалярное произведение векторов *vector_a* и *vector_b*, равное сумме произведений их элементов с равными значениями индексов.

vector_a - одномерный массив целого, вещественного, комплексного или логического типа.

vector_b - одномерный массив того же размера, что и массив *vector_a*. Должен быть логического типа, если массив *vector_a* логического типа. Должен быть числовым (целым, вещественным или комплексным), если массив *vector_a* числовой. В последнем случае тип *vector_b* может отличаться от типа *vector_a*.

Возвращаемое число равно:

- SUM(*vector_a* * *vector_b*), если *vector_a* целого или вещественного типа. Результат имеет целый тип, если оба аргумента целого типа; комплексный, если *vector_b* комплексного типа, и вещественный в противном случае;
- SUM(CONJG(*vector_a*) * *vector_b*), если *vector_a* комплексного типа, то и результат также является комплексным числом;
- ANY(*vector_a* .AND. *vector_b*), если аргументы логического типа, то и результат имеет логический тип.

Если размер векторов равен нулю, то и результат равен нулю или .FALSE. в случае логического типа.

Пример:

```
print *, dot_product((/ 1, 2, 3 /), (/ 4, 5, 6 /))      !   32
```

MATMUL(*matrix_a*, *matrix_b*) - выполняет по принятым в линейной алгебре правилам умножение матриц целого, вещественного, комплексного и логического типа.

matrix_a - одномерный или двумерный массив целого, вещественного, комплексного или логического типа.

matrix_b – массив логического типа, если *matrix_a* - логический массив; числовой массив, если *matrix_a* - числовой массив. В последнем случае тип *matrix _b* может отличаться от типа *matrix_a*.

По крайней мере один из массивов *matrix_a* и *matrix_b* должен быть двумерным.

Возможны 3 случая:

- *matrix_a* имеет форму (*n*, *m*), а *matrix_b* имеет форму (*m*, *k*). Тогда результат имеет форму (*n*, *k*), а значение элемента (*i*, *j*) равно SUM(*matrix_a*(*i,* :) * *matrix_b*(:, *j*));
- *matrix_a* имеет форму (*m*), а *matrix_b* имеет форму (*m*, *k*). Тогда результат имеет форму (*k*), а значение элемента (*j*) равно SUM(*matrix_a* * *matrix_b*(:, *j*));

- *matrix_a* имеет форму (*n*, *m*), а *matrix_b* имеет форму (*m*). Тогда результат имеет форму (*n*), а значение элемента (*i*) равно SUM(*matrix_a*(*i,* :) * *matrix_b*).

Для логических массивов функция ANY эквивалентна функции SUM, а .AND. эквивалентен произведению (*).

Пример:

```
integer a(2, 3), b(3, 2), c(2), d(3), e(2, 2), f(3), g(2)
a = reshape((/ 1, 2, 3, 4, 5, 6 /), (/ 2, 3 /))
b = reshape((/ 1, 2, 3, 4, 5, 6 /), (/ 3, 2 /))
! Массив a:    1    3    5
!              2    4    6
! Массив b:    1    4
!              2    5
!              3    6
c = (/ 1, 2 /)
d = (/ 1, 2, 3 /)
e = matmul(a, b)              ! Результат: 22   49
                              !            28   64
f = matmul(c, a)              ! Результат:  5   11   17
g = matmul(a, d)              ! Результат: 22   28
```

Из линейной алгебры известно, что произведением вектор-столбца x на вектор-строку y^T является матрица

$$A = \begin{pmatrix} x_1 \\ x_2 \\ \dots \\ x_m \end{pmatrix} \begin{pmatrix} y_1 & y_2 \dots y_m \end{pmatrix} = xy^T \in \mathbf{R}^{m \times n} .$$

В Фортране ее вернет встроенная функция MATMUL. Правда, для обеспечения ее работоспособности потребуется преобразовать вектор x в массив формы (/ *m*, 1 /), а вектор y^T - в массив формы (/ 1, *n* /):

```
integer(4), parameter :: m = 3, n = 5
integer(4) :: i, j
real(4) :: x(m) = (/ 1.0, 2.0, 3.0 /)
real(4) :: y(n) = (/ 5.0, 6.0, 7.0, 8.0, 9.0 /)
real(4) :: a(m, n) = 0.0
a = matmul(reshape(x, shape = (/ m, 1 /)), reshape(y, shape = (/ 1, n /)))
do i = 1, m                   ! Вывод результата
  print '(1x, 10f6.2)', a(i, :)
end do
! Тот же результат, но быстрее, даст вложенный цикл
do j = 1, n
 do i = 1, m
  a(i, j) = x(i) * y(j)
 end do
end do
```

Результат:

```
 5.00   6.00   7.00   8.00   9.00
10.00  12.00  14.00  16.00  18.00
15.00  18.00  21.00  24.00  27.00
```

Замечание. Рассмотренное умножение вектор-столбца *x* на вектор-строку y^T называют *внешним произведением*. Ранг внешнего произведения, когда *x* и *y* действительные векторы, не выше единицы.

4.12.3. Справочные функции для массивов

4.12.3.1. Статус размещаемого массива

ALLOCATED(*array*) - возвращает значение стандартного логического типа, равное .TRUE., если размещаемый массив *array* (массив, имеющий атрибут ALLOCATABLE) в данный момент размещен, и .FALSE. - в противном случае. Результат будет неопределенным, если не определен статус размещаемого массива. Возникнет ошибка компиляции, если параметром функции окажется массив, не имеющий атрибут ALLOCATABLE, или скаляр.

4.12.3.2. Граница, форма и размер массива

Функции этого раздела выдают информацию о границах массива любого типа. Если параметром является размещаемый массив, то он должен быть размещен, а если ссылка, то она должна быть прикреплена к адресату. Нижние границы сечения массива считаются равными единице, а верхние - равными соответствующим экстентам. Поскольку результат зависит только от свойств массива, то его значение необязательно должно быть определенным. В функциях этого подраздела (кроме функции SHAPE) параметр *dim* - целое константное выражение; $1 \le dim \le n$, где *n* - ранг массива - аргумента функции.

LBOUND(*array* [, *dim*]) - если параметр *dim* отсутствует, то возвращается одномерный массив стандартного целого типа, содержащий нижние границы всех измерений. Размерность массива-результата при отсутствии *dim* равна рангу массива *array*. Если *dim* задан, то результатом является скаляр, равный нижней границе размерности *dim* массива *array*.

Если *array* - перенимающий размер массив, то параметр *dim* должен быть задан и не должен задавать последнюю размерность массива *array*.

dim - целочисленное константное выражение; $1 \le dim \le n$, где *n* - ранг массива *array*.

UBOUND(*array* [, *dim*]) - подобна LBOUND, но возвращает верхние границы.

Пример:

```
real array (2:8, 8:14)
integer, allocatable :: lb(:)
allocate( lb(size(shape(array))) )
```

```
lb = lbound(array)
print *, lb                          !      2       8
print *, lbound(array, dim = 2)      !      8
print *, lbound(array(2:6:2, 10:12)) !      1       1
lb = ubound(array)
print *, lb                          !      8      14
print *, ubound(array, dim = 2)      !     14
print *, ubound(array(:6:2, 10:12))  !      3       3
end
```

SHAPE(*source*) - возвращает одномерный массив стандартного целого типа, содержащий форму массива или скаляра *source*. *Source* может иметь любой тип и не может быть перенимающим размер массивом. Размер массива-результата равен рангу *source*.

Пример:

```
integer vec(2), array(3:10, -1:3)
vec = shape(array)
write(*, *) vec                      !      8       5
```

SIZE(*array* [, *dim*]) - возвращает стандартное целое, равное размеру массива *array*, или, если присутствует скалярный целый параметр *dim*, число элементов (экстент) вдоль заданного измерения *dim*. Если *array* - перенимающий размер массив, параметр *dim* должен быть задан.

Пример:

```
real(8) array (3:10, -1:3)
integer i, j
i = size(array, dim = 2)             ! Возвращает 5
j = size(array)                      ! Возвращает 40
```

4.12.4. Функции преобразования массивов

4.12.4.1. Элементная функция MERGE слияния массивов

MERGE(*tsource*, *fsource*, *mask*) - создает согласно заданной маске новый массив из элементов двух массивов.

tsource, *fsource* - массивы одной формы, одного (любого) типа и параметра типа, из которых берутся элементы в массив-результат.

mask - логический массив той же формы, которую имеют массивы *tsource* и *fsource*. Массив *mask* определяет, из какого массива, *tsource* или *fsource*, будет взят в массив-результат очередной элемент.

Функция MERGE возвращает массив той же формы и того же типа, что и массивы *tsource* и *fsource*. В массив-результат поступает элемент массива *tsource*, если соответствующий ему элемент в массиве *mask* равен .TRUE., в противном случае в результат поступает элемент из массива *fsource*.

Пример:

```
integer tsource(2, 3), fsource(2, 3), ar1 (2, 3)
logical mask(2, 3)
tsource = reshape((/1, 4, 2, 5, 3, 6/), (/2, 3/))
fsource = reshape((/7, 0, 8, -1, 9, -2/), (/2, 3/))
mask = reshape((/.true., .false., .false., .true., .true., .false./), (/2,3/))
! tsource:    1   2   3  fsource:    7   8   9  mask:      .true.  .false.  .true.
!            4   5   6               0  -1  -2             .false.  .true.  .false.
ar1 = merge(tsource, fsource, mask)        ! Результат:  1       8       3
end                                        !             0       5      -2
```

Замечание. Параметр *tsource* или *fsource* может быть и скаляром, который по правилам элементности будет расширен в массив надлежащей формы, например:

```
integer tsource(5) / 1, 2, 3, 4, 5 /, fsource / 7 /
logical mask(5) / .true., .false., .false., .true., .true. /
print *, merge(tsource, fsource, mask)        !   1   7   7   4   5
end
```

4.12.4.2. Упаковка и распаковка массивов

PACK(*array*, *mask* [, *vector*]) - упаковывает массив в одномерный массив (вектор) под управлением массива *mask*.

array - массив любого типа, который пакуется в вектор. *mask* - логический массив той же формы, которую имеет и *array*, или просто логическая величина .TRUE.; *mask* - задает условия упаковки элементов массива *array*.

vector - необязательный одномерный массив, имеющий тот же тип и разновидность типа, что и массив *array*. Число элементов в массиве не должно быть меньше количества элементов со значением .TRUE. в массиве *mask*.

Функция возвращает одномерный массив того же типа и разновидности типа, что и массив *array*, и того же размера, что и массив *vector*, если последний задан. Значение первого элемента в массиве-результате - элемент массива *array*, который соответствует элементу со значением .TRUE. в *mask*; второй элемент в массиве-результате - элемент массива *array*, который соответствует второму элементу со значением .TRUE. в *mask*, и т. д. Элементы просматриваются в порядке их размещения в памяти ЭВМ (быстрее всего изменяется самый левый индекс). Если *vector* опущен, то размер результирующего массива равен числу элементов со значением .TRUE. в *mask*. Если же параметр *mask* задан единственным значением .TRUE., то размер результата равен размеру массива *array*. Если *vector* задан и имеет размер, больший числа элементов со значением .TRUE. в *mask*, то дополнительные элементы массива *vector* копируются без изменений в результат.

Пример:

```
integer array(2, 3), vec1(2), vec2(5)
logical mask (2, 3)
array = reshape((/ 7, 0, 0, -5, 0, 0 /), (/ 2, 3 /))
mask = array /= 0
! Массив array:      7     0     0    Массив mask:        . true.    . false.    . false.
!                    0    -5     0                        .false.     .true.    . false.

vec1 = pack(array, mask)
vec2 = pack(array, mask = array > 0, vector = (/ 1, 2, 3, 4, 5 /))
print *, vec1                         !     7     -5
print *, vec2                         !     7     2     3     4     5
end
```

UNPACK(*vector*, *mask*, *field*) - возвращает массив того же типа и разновидности типа, как и у одномерного массива *vector*, и той же формы, что у логического массива *mask*. Число элементов *vector* по меньшей мере равно числу истинных элементов массива *mask*. Параметр *field* должен быть скаляром либо иметь ту же форму, которую имеет и массив *mask*, а его тип и параметры типа должны быть такими же, как у *vector*.

Элемент результата, соответствующий *i*-му истинному элементу массива *mask*, считая в порядке следования его элементов, равен *i*-му элементу *vector*, а все остальные элементы равны соответствующим элементам *field*, если это массив, или собственно *field*, если это скаляр.

Пример:

```
logical mask (2, 3)
integer vector(3) /1, 2, 3/, ar1(2, 3)
mask = reshape((/ .true., .false., .false., .true., .true., .false. /), (/ 2, 3 /))
! Массив vector:    1     2     3    Массив mask:        . true.    . false.    . true.
!                                                        .false.     .true.    . false.

ar1 = unpack(vector, mask, 8)
print *, ar1(1, :)                    ! Результат:        1          8         3
print *, ar1(2, :)                    !                   8          2         8
end
```

4.12.4.3. Переформирование массива

RESHAPE(*source*, *shape* [, *pad*] [, *order*]) - формирует массив заданной формы *shape* из элементов массива *source*. Результирующий массив имеет тот же тип и разновидность типа, что и *source*.

source - массив любого типа, элементы которого берутся в порядке их следования для формирования нового массива.

shape - одномерный целочисленный массив, задающий форму результата: *i*-й элемент *shape* равен размеру *i*-го измерения формируемого массива. Если *pad* опущен, общий задаваемый *shape* размер не должен превышать размера *source*.

pad - необязательный массив того же типа, что и *source*. Если в *source* недостает элементов для формирования результата, элементы *pad* добавляются в результирующий массив в порядке их следования. При необходимости используются дополнительные копии *pad* для заполнения результата.

order - необязательный одномерный массив того же размера, что и *shape*. Переставляет порядок измерений (что изменяет порядок заполнения) массива-результата. Значениями *order* должна быть одна из перестановок вида (1, 2, ..., *n*), где *n* - размер *shape*; *order* задает порядок изменения индексов при заполнении результата. Быстрее всего изменяется индекс *order*(1), медленнее всего - *order*(*n*). При этом элементы из *source* выбираются в нормальном порядке. Далее при нехватке элементов *source* следуют копии элементов *pad*. Параметр *order* позволяет, в частности, переформировывать массивы в принятом в СИ порядке с последующей их передачей в СИ-функцию.

Пример:

```
integer ar1(2, 5)
real f(5,3,8), c(8,3,5)
ar1 = reshape((/ 1, 2, 3, 4, 5, 6 /), (/ 2, 5 /), (/ 0, 0 /), (/ 2, 1 /))
print *, ar1(1, :)
print *, ar1(2, :)
! Результат:      1    2    3    4    5
!                 6    0    1    0    1
! Изменим принятый в Фортране порядок на порядок, принятый в СИ
c = reshape(f, (/ 8, 3, 5 /), order = (/ 3, 2, 1 /))
```

4.12.4.4. Построение массива из копий исходного массива

SPREAD(*source*, *dim*, *ncopies*) - повторяет массив *source* вдоль заданного измерения в массиве-результате, ранг которого на единицу больше *source*.

source - массив или скалярная величина любого типа.

dim - целый скаляр, задающий измерение, вдоль которого будет повторен *source*; $1 \le dim \le n+1$, где *n* - число измерений в *source*.

ncopies - число повторений *source*; равняется размеру экстента добавляемого измерения.

Функция возвращает массив того же типа и разновидности типа, что и у *source*. Если *source* скаляр, то элемент результата равен собственно *source*. Результат содержит MAX(*ncopies*, 0) копий *source*.

Пример:

```
integer ar1(2, 3), ar2(3, 2)
ar1 = spread((/ 1, 2, 3 /), dim=1, ncopies=2)   ! Результат:    1    2    3
                                                !               1    2    3
ar2 = spread((/ 1, 2, 3 /), 2, 2)               ! Результат:    1    1
```

```
!                    2    2
!                    3    3
```

4.12.4.5. Функции сдвига массива

CSHIFT(*array*, *shift* [, *dim*]) - выполняет циклический сдвиг массива *array* по заданному необязательному индексу *dim*.

array - массив любого типа.

shift - число позиций (INTEGER), на которое сдвигаются элементы *array*. Может быть целочисленным массивом, ранг которого на единицу меньше ранга *array*. Если *shift* - скаляр, то результат получается циклическим сдвигом каждого одномерного сечения по индексу *dim* на *shift* позиций. Если *shift* - массив, то каждый его элемент задает сдвиг для соответствующего сечения *array*. При этом форма *shift* должна совпадать с формой *array* за вычетом размерности *dim*. Положительный сдвиг выполняется в направлении уменьшения индексов (влево в случае вектора), и, наоборот, отрицательный сдвиг выполняется в направлении увеличения значений индексов массива (вправо в случае вектора).

dim - необязательный параметр (INTEGER), задающий индекс, по которому выполняется сдвиг; $1 \leq dim \leq n$, где n - ранг *array*. Если *dim* опущен, то сдвиг выполняется по первому индексу.

Функция возвращает массив, в котором выполнен циклический сдвиг элементов, того же типа и формы, как и у *array*. Если ранг *array* больше единицы, то циклически сдвигается каждое одномерное сечение по заданному индексу *dim* или по первому индексу, если *dim* опущен.

Пример:

```
integer array (3, 3), ar1(3, 3), ar2 (3, 3)
data  array / 1, 4, 7, 2, 5, 8, 3, 6, 9 /
! Массив array:      1    2    3
!                    4    5    6
!                    7    8    9

! Сдвиг в каждом столбце на одну позицию
ar1 = cshift(array, 1, dim = 1)
! Результат:         4    5    6
!                    7    8    9
!                    1    2    3

! Сдвиг в первом ряду на -1, во втором - на 1
ar2=cshift(array, shift=(/-1, 1, 0/), dim = 2)


! Результат:         3    1    2
!                    5    6    4
!                    7    8    9
```

EOSHIFT(*array*, *shift* [, *boundary*] [, *dim*]) - выполняет вытесняющий левый или правый сдвиг по заданному необязательному индексу *dim* и

заполняет необязательными краевыми значениями образуемые в результате сдвига пропуски.

array - массив любого типа.

shift, *dim* - имеют тот же смысл, что и для функции CSHIFT.

boundary - необязательный параметр того же типа, что и *array*. Задает значения, которыми заполняются возникающие в результате сдвига пропуски. Может быть массивом граничных значений, ранг которого на единицу меньше ранга *array*. Если *boundary* опущен, то задаваемые по умолчанию замены зависят от типа *array*: целый - 0; вещественный - 0.0; комплексный - (0.0, 0.0); логический - .FALSE.; символьный - пробел.

Функция возвращает массив, в котором выполнены сдвиг и замены.

Пример:

```
integer shift(3)
character(1) array(3, 3), ar1(3, 3)
array = reshape((/'a', 'd', 'g', 'b', 'e', 'h', 'c', 'f', 'i'/), (/3, 3/))
! Массив array:      a     b     c
!                    d     e     f
!                    g     h     i
shift = (/-1, 1, 0/)
ar1 = eoshift (array, shift, boundary = (/'*', '?', '#'/), dim = 2)
! Результат:        *     a     b
!                    e     f     ?
!                    g     h     i
```

4.12.4.6. Транспонирование матрицы

TRANSPOSE(*matrix*) - меняет местами (транспонирует) столбцы и строки матрицы (двумерного массива) *matrix*. Тип и разновидность типа результирующего массива и *matrix* одинаковы. Если *matrix* имеет форму (*k*, *n*), то результат имеет форму (*n*, *k*).

Пример:

```
integer array(2, 3), result(3, 2)
array = reshape((/1, 2, 3, 4, 5, 6/), (/2, 3/))
! Массив array:      1     3     5
!                    2     4     6
result = transpose(array)
! Результат:        1     2
!                    3     4
!                    5     6
```

4.13. Ввод/вывод массива под управлением списка

Управляемый списком В/В используется при работе с последовательными текстовыми файлами и стандартными устройствами

(клавиатура, экран, принтер). Преобразования В/В выполняются в соответствии с типами и значениями вводимых и выводимых величин.

При вводе массива из файла возможны случаи:

- известно число вводимых данных;
- необходимо ввести весь файл, но его размер до ввода неизвестен.

В последнем случае правильнее выполнять ввод в динамический массив.

4.13.1. Ввод/вывод одномерного массива

Рассмотрим пример В/В одномерного статического массива.

```
integer, parameter :: nmax = 20
integer :: ios, n = 15, i              ! Планируем ввести n значений
character(60) :: fn = 'a.txt'
real :: a(nmax) = 0
open(2, file = fn, status = 'old', iostat = ios)
if(ios /= 0) then                      ! Останов в случае ошибки
 print *, 'Не могу открыть файл ' // trim(fn)
 stop
end if
if(n > nmax) stop 'Размер списка ввода больше размера массива'
read(2, *, iostat = ios) (a(i), i = 1, n)
if(ios /= 0) then
 write(*, *) 'Число введенных данных меньше заявленного.'
 n = i - 1                             ! n - число введенных значений
 if(eof(2)) backspace 2                ! Позиционируем файл перед
end if                                 ! записью "конец файла"
write(2, *, iostat = ios) (a(i), i = 1, n)
close(2)                               ! Закрываем файл
write(*,'(1x, 5f5.2)') a(1:n)          ! Контрольный вывод на экран
end
```

Состав файла a.txt (до выполнения оператора WRITE(2, *, ...):

```
1.0  2.0  3.0
4.0  5.0
```

Отображаемый на экране результат:

```
Число введенных данных меньше заявленного.
1.00  2.00  3.00  4.00  5.00
```

Пояснения:

1. Оператор OPEN открывает устройство В/В с номером 2 и подсоединяет к нему файл a.txt. При удачном подсоединении файл a.txt открыт. Далее в программе для доступа к файлу используется номер устройства.

Параметр *status* = '*old*' означает, что открываемый файл должен существовать. Параметр *ios* позволяет передать в программу код завершения выполнения оператора OPEN. Целочисленная переменная *ios*

равна нулю при успешном открытии файла и отлична от нуля, если возникла ошибка.

После подсоединения файл позиционируется в свое начало. Файл a.txt открывается и для чтения и для записи. Доступ к файлу последовательный.

2. Контроль ввода выполняется параметром *ios*: если нет ошибок ввода, то значение *ios* равно нулю; если достигнут конец файла, значение *ios* равно -1; *ios* больше нуля, если имели место иные ошибки. В нашем примере будет достигнут конец файла, однако параметр *i* циклического списка оператора READ сохраняет свое значение после завершения работы READ, что позволит подсчитать число введенных элементов. Правда, такой способ определения числа введенных элементов не сработает, если ввод данных будет прекращен при обнаружении в файле a.txt слеша (/), поскольку в этом случае *ios* = 0.

3. Оператор ввода содержит циклический список ($a(i)$, $i = 1, n$). Это позволяет прочитать первые n произвольно размещенных полей (если, конечно, не возникло ошибки ввода). Ввод выполняется с начала записи, и если оператор ввода должен прочитать больше полей, чем находится в текущей записи, то недостающие поля будут взяты из последующих записей. Каждый оператор READ (если не задан ввод без продвижения) начинает ввод с начала новой записи файла. Поэтому в случае применения цикла

```
do i = 1, n
 read(2, *, iostat = ios) a(i)
end do
```

нам потребовалось бы расположить в текстовом файле каждое число на отдельной строке, т. е. в столбик, что выглядит весьма неуклюже.

С появлением сечений циклический список ($a(i)$, $i = 1, n$) можно заменить сечением $a(1{:}n)$. В нашем примере сечение применено в операторе WRITE.

Для ввода также можно применить оператор:

```
read(2, *, iostat = ios) a              ! или a(1:nmax)
```

Он попытается ввести весь массив - передать из файла первые *nmax* полей. Однако если полей ввода меньше *nmax*, то при таком вводе уже нельзя вычислить число введенных данных.

Вывод всего массива выполнит оператор

```
write(*, *) a                           ! Вывод на экран
```

4. Вывод массива в файл a.txt приведет к тому, что в файл начиная с новой строки (записи) будут переданы n элементов массива a. Строка, начиная с которой будут передаваться данные, определяется по правилу: если файл установлен на строке *line*, то новые данные будут добавляться начиная со строки $line + 1$. Причем, поскольку доступ к файлу

последовательный, все существующие после строки *line* данные будут "затерты" (заменены на вновь выводимые). В общем случае при управляемом списком выводе *каждый оператор вывода* создает одну запись, если длина создаваемой записи не превышает 79 символов. Число полей в созданной записи равно числу элементов в списке вывода. Если же для размещения элементов вывода требуется большее число символов, то создаются новые записи. В качестве разделителей между полями оператор WRITE использует пробелы.

5. Оператор CLOSE(2) закрывает файл a.txt - отсоединяет файл от устройства 2.

Рассмотрим теперь, как ввести весь файл или все его первые числовые поля в размещаемый массив. Прежде следует подсчитать, сколько данных можно ввести из файла, затем выделить под массив память, "перемотать" файл в его начало и ввести в массив данные. Например:

```
integer, parameter :: nmax = 10000
integer, parameter :: unt = 3
integer :: ios, i, n
character(60) :: fn='a.txt'          ! Используем файл предыдущего примера
real tmp                             ! Используем tmp для подсчета количества
real, allocatable :: a(:)            ! подряд идущих чисел в файле;
                                     ! tmp имеет тот же тип, что и массив a
open(unt, file = fn, status = 'old')
read(unt, *, iostat = ios) (tmp, i = 1, nmax)
if(ios == 0)  stop 'Нельзя ввести весь файл'
n = i - 1                            ! n - число вводимых данных
allocate( a(n) )                     ! Выделяем память под массив a
rewind unt                           ! Переход в начало файла
read(unt, *) a                       ! Ввод массива a
close(unt)                           ! Закрываем файл
write(*, '(1x, 5f5.2)') a(:n)
deallocate(a)
end
```

4.13.2. Ввод/вывод двумерного массива

Пусть надо ввести из файла b.txt данные в двумерный массив *a*(1:3, 1:4). Занесем в файл b.txt данные так, чтобы строка файла соответствовала одной строке массива *a*(1:3, 1:4):

```
11  12  13  14
21  22  23  24
31  32  33  34
```

Тогда ввод массива по строчкам можно выполнить в цикле

```
do i = 1, 3
 read(2, *, iostat = ios) (a(i, j), j = 1, 4)          ! или a(i, 1:4)
end do
```

Для ввода *i*-й строки массива вновь использован циклический список.

В принципе при управляемом списком вводе двумерного массива можно использовать и вложенный циклический список:

```
read(2, *, iostat = ios) ((a(i, j), j = 1, 4), i = 1, 3)
```

Такой оператор также обеспечит ввод данных в массив *a* построчно (быстрее изменяется параметр *j*). Однако при этом вводимые данные не обязательно располагать в трех строчках, по 4 числа в каждой. Они могут быть размещены, например, так:

```
11 12 13 14   21 22 23 24
31 32 33 34
```

Если же расположение данных в файле соответствует их размещению в столбцах массива (т. е. их порядок в файле совпадает с порядком их размещения в памяти ЭВМ), например так:

```
11 21 31
12 22 32
13 23 33
14 24 34
```

или так:

```
11 21 31      12 22 32      13 23 33      14 24 34
```

то ввод *всего* массива можно выполнить оператором

```
read(2, *, iostat = ios) a
```

Контрольный вывод массива на экран по строкам организуется в цикле

```
do i = 1, 3
  write(*, *) (a(i, j), j = 1, 4)        ! или a(i, 1:4)
end do
```

Замечание. В рассмотренных нами примерах к файлам был организован последовательный метод доступа, при котором возможно лишь чтение и запись данных. Редактирование отдельных записей файла становится возможным при прямом методе доступа (гл. 10).

5. Выражения, операции и присваивание

В настоящей главе обобщаются сведения о выражениях и операциях. Операции применяются для создания выражений, которые затем используются в операторах Фортрана.

Операции Фортрана разделяются на встроенные и задаваемые программистом (перегружаемые).

Встроенные операции:

- арифметические;
- символьная операция конкатенации (объединение символьных строк);
- операции отношения;
- логические.

Символьные выражения и операция конкатенации в этой главе не рассматриваются, поскольку подробно изложены в разд. 3.8.5.

Часть арифметических действий реализована в Фортране в виде встроенных функций, например вычисление остатка от деления, усечение и округление, побитовые операции и др. Рассмотрение встроенных функций выполнено в следующей главе.

Выражения подразделяются на скалярные и выражения-массивы. Результатом выражения-массива является массив или его сечение. По крайней мере одним из операндов выражения-массива должны быть массив или сечение массива, например:

```
real :: a(5) = (/ (i, i = 1, 5) /)
a(3:5) = a(1:3) * 2                ! Возвращает (1.0  2.0  2.0  4.0  6.0)
```

5.1. Арифметические выражения

Результатом арифметического выражения может быть величина целого, или вещественного, или комплексного типа или массив (сечение) одного из этих типов. Операндами арифметического выражения могут быть:

- арифметические константы;
- скалярные числовые переменные;
- числовые массивы и их сечения;
- вызовы функций целого, вещественного и комплексного типа.

5.1.1. Выполнение арифметических операций

Арифметические операции различаются приоритетом:

** возведение в степень (операция с наивысшим приоритетом);

*, / умножение, деление;

одноместные (унарные) + и -;

+, - сложение, вычитание.

Замечание. В Фортране в отличие от СИ унарным операциям не может предшествовать знак другой операции, например:

```
k = 12 / -a                    ! Ошибка
k = 12 / ( -a )                ! Правильно
```

Операции Фортрана, кроме возведения в степень, выполняются *слева направо* в соответствии с приоритетом. Операции возведения в степень выполняются *справа налево*. Так, выражение -a + b + c будет выполнено в следующем порядке: (((-a) + b) + c). А выражение a**b**c вычисляется так: (a**(b**c)). Заключенные в круглые скобки подвыражения вычисляются в первую очередь.

Пример:

```
k = 2 * 2 ** 2 / 2 / 2          ! 2
! Проиллюстрируем последовательность вычислений, расставив скобки:
k = ((2 * (2 ** 2)) / 2) / 2    ! 2
! Проиллюстрируем влияние скобок на результат выражения
k = 2 ** 8 / 2 + 2              ! 130
k = 2 ** (8 / 2 + 2)            ! 64
k = 2 ** (8 / (2 + 2))          ! 4
```

В арифметических выражениях запрещается:

- делить на нуль;
- возводить равный нулю операнд в отрицательную или нулевую степень;
- возводить отрицательный операнд в нецелочисленную степень.

Пример:

```
a = (-2)**2.2                   ! Ошибка - нарушение последнего ограничения
```

5.1.2. Целочисленное деление

Рассмотрим простую программу:

```
real(4) dp, dn
dp =  3 / 2
dn = -3 / 2
print *, dp, dn                 !     1.0     -1.0
end
```

Программисты, впервые наблюдающие целочисленное деление, будут удивлены, увидев в качестве результата 1.0 и -1.0 вместо ожидаемых ими 1.5 и -1.5. Однако результат имеет простое объяснение: 3, -3 и 2 - целые числа и результатом деления будут также целые числа - целая часть числа 1.5 и целая часть числа -1.5, т. е. 1 и -1. Затем, поскольку переменные *dp* и *dn* имеют тип REAL(4), целые числа 1 и -1 будут преобразованы в стандартный вещественный тип.

Чтобы получить ожидаемый с позиции обычной арифметики результат, в программе можно записать:

dp = 2.0/3.0 или dp = 2/3.0, или dp = 2.0/3

Можно также воспользоваться функциями явного преобразования целого типа данных в вещественный (разд. 6.5) и записать, например,

d = float(2)/float(3) или d = real(2)/real(3)

Еще несколько примеров целочисленного деления:

2 ** (-2) возвращает 0 (целочисленное деление);

2.0 ** (-2) возвращает 0.25 (нет целочисленного деления);

-7/3 возвращает -2;

19/10 возвращает 1;

1/4 + 1/4 возвращает 0.

5.1.3. Ранг и типы арифметических операндов

В Фортране допускается использовать в арифметическом выражении операнды разных типов и разновидностей типов. В таком случае результат каждой операции выражения определяется по следующим правилам:

- если операнды арифметической операции имеют один и тот же тип, то результат операции имеет тот же тип. Это правило хорошо иллюстрируется целочисленным делением;
- если операнды операции имеют различный тип, то результат операции имеет тип операнда наивысшего ранга.

Ранг типов арифметических операндов (дан в порядке убывания):

COMPLEX (8) или DOUBLE COMPLEX - наивысший ранг;

COMPLEX (4);

REAL(8) или DOUBLE PRECISION;

REAL(4) или REAL;

INTEGER(4) или INTEGER;

INTEGER(2);

INTEGER(1) или BYTE - низший ранг.

Пример:

```
integer(2) :: a = 1, b = 2
real(4) :: c = 2.5
real(8) d1, d2
d1 = a / b * c                  ! 0.0_8
d2 = a / (b * c)                ! 0.2_8
```

При вычислении *d*2 первоначально выполняется операция умножения, но прежде число 2 типа INTEGER(2) переводится в тип REAL(4). Далее

выполняется операция деления, и вновь ее предваряет преобразование типов: число 1 типа INTEGER(2) переводится в тип REAL(4). Выражение возвращает 0.2 типа REAL(4), которое, однако, в результате присваивания преобразовывается в тип REAL(8). При этом типы операндов выражения - переменных *a* и *b*, разумеется, сохраняются.

В ряде случаев в выражениях, например вещественного типа с целочисленными операндами, чтобы избежать целочисленного деления, следует выполнять явное преобразование типов данных, например:

```
integer :: a = 8, b = 3
real c
c = 2.0**(a / b)                    ! 4.0
c = 2.0**(float(a) / float(b))      ! 6.349604
```

Встроенные математические функции, обладающие специфическими именами, требуют точного задания типа аргумента, например:

```
real(4) :: a = 4
real(8) :: b = 4, x
x = dsqrt(a)                ! Ошибка: тип параметра должен быть REAL(8)
x = dsqrt(b)                ! Верно
```

Перевод числа к большей разновидности типа, например от REAL(4) к REAL(8), может привести к искажению точности, например:

```
real(4) :: a = 1.11
real(8) :: c
c = a
print *, a                  !       1.110000
print *, c                  !       1.110000014305115
```

В то же время если сразу начать работу с типом REAL(8), то точность сохраняется, например:

```
real(8) :: c
c = 1.11_8                  ! или c = 1.11d0
print *, c                  !       1.110000000000000
```

Искажение значения может произойти и при переходе к низшей разновидности типа, например:

```
integer(2) :: k2 = 325
integer(1) :: k1            ! -128 <= k1 <= 127
k1 = k2
print *, k2                 !      325
print *, k1                 !       69
```

5.1.4. Ошибки округления

Необходимо учитывать, что арифметические выражения с вещественными и комплексными операндами вычисляются неточно, т. е. при их вычислении возникает *ошибка округления*. В ряде случаев

пренебрежение такой ошибкой приводит к созданию неработоспособных программ, например следующий цикл является бесконечным, поскольку *x* из-за ошибки округления не принимает значения, точно равного 1.0.

Пример:

```
real :: x = 0.1
do                                  ! Бесконечный цикл
 print *, x
 x = x + 0.1
 if(x == 1.0) exit                  ! EXIT - оператор выхода из цикла
end do
```

Нормальное завершение цикла можно обеспечить так:

```
real :: x = 0.1
do
 print *, x
 x = x + 0.1
 if(abs(x - 1.0) < 1.0e-5) exit     ! x практически равен 1.0
end do                              ! ABS(x - 1.0) возвращает |x - 1.0|
```

Общий вывод из приведенного примера: нельзя сравнивать вещественные числа на предмет точного равенства или неравенства, а следует выполнять их сравнение с некоторой точностью.

Влияние ошибок округления можно снизить, правильно формируя порядок вычислений. Пусть, например, объявлены и инициализированы переменные *x*, *y* и *z*:

```
real(4) :: x = 1.0e+30, y = -1.0e+30, z = 5.0
```

Их сумма равна 5.0. Однако найдем и выведем их сумму так:

```
print *, x + (y + z)                !     0.000000E+00     (ошибка)
```

Результат ошибочен. Правильной является такая последовательность вычислений:

```
print *, (x + y) + z                !     5.000000         (верно)
```

5.2. Выражения отношения и логические выражения

Выражение отношения сравнивает значения двух арифметических или символьных выражений. Арифметическое выражение можно сравнить с символьным выражением. При этом арифметическое выражение рассматривается как символьное - последовательность байтов. Результатом выражения отношения является .TRUE. или .FALSE..

Операндами выражения отношения могут быть как скаляры, так и массивы или их сечения, например:

```
(/ 1, 2, 3 /) > (/ 0, 3, 0 /)       ! Возвращает массив (T  F  T)
```

Операции отношения могут быть записаны в двух формах:

.LT. или < меньше;

.LE. или <= меньше или равно;

.GT. или > больше;

.GE. или >= больше или равно;

.EQ. или == равно;

.NE. или /= не равно.

Пробелы в записи обозначения операции являются ошибкой:

```
a . le. b                    ! Ошибка. Правильно: a .le. b
a < = b                      ! Ошибка. Правильно: a <= b
```

Все операции отношения являются двуместными (бинарными) и должны появляться между операндами. Выполняются операции отношения слева направо.

Если в выражении отношения один операнд имеет вещественный, а другой целый тип, то перед выполнением операции целочисленный операнд преобразовывается в вещественный тип.

Выражения отношения с символьными операндами сравниваются посимвольно. Фактически выполняется сравнение кодов символов сравниваемых строк. При сравнении строк разной длины короткая строка увеличивается до длины большей строки за счет добавления завершающих пробелов, например выражение 'Expression' > 'Exp1' вычисляется как 'Expression' > 'Exp1□□□□□□' (здесь символ □ обозначает пробел).

Операнды выражения отношения могут иметь и комплексный тип. В этом случае можно применять только операции .NE. (/=) и .EQ. (==).

Логические выражения имеют результатом логическое значение *истина* - .TRUE. или *ложь* - .FALSE.. Операндами логических выражений могут быть:

- логические константы, переменные и функции;
- массивы логического и целого типа и их сечения;
- выражения отношения;
- целочисленные константы, переменные и функции.

Логические операции:

.NOT. логическое НЕ (отрицание);

.AND. логическое И;

.OR. логическое ИЛИ;

.XOR. логическое исключающее ИЛИ;

.EQV. эквивалентность;

.NEQV. неэквивалентность.

Все логические операции, кроме отрицания, являются бинарными. Логическая операция .NOT. является унарной и располагается перед операндом. Выполняются логические операции слева направо.

В табл. 5.1 приведены результаты логических операций над логическими переменными *x* и *y*, принимающими значения *истина* (И) и *ложь* (Л).

Таблица 5.1. Таблица истинности

x	*y*	*x* .AND. *y*	*x* .OR. *y*	.NOT. *x*	*x* .XOR. *y*	*x* .EQV. *y*	*x* .NEQV. *y*
И	И	И	И	Л	Л	И	Л
И	Л	Л	И	Л	И	Л	И
Л	И	Л	И	И	И	Л	И
Л	Л	Л	Л	И	Л	И	Л

Операнды логических операций должны быть логического типа. Однако CVF и FPS также допускают использование операндов целого типа. В этом случае логические операции выполняются побитово. Если операнды имеют различные разновидности целого типа, то выполняется преобразование типов - операнд целого типа меньшего ранга преобразовывается в целый тип наибольшего ранга. Логическое выражение с целочисленными операндами имеет результат *целого*, а не *логического* типа, например:

```
write(*, *) 2#1000 .or. 2#0001        !    9      ( = 1001 )
write(*, *) 8 .or. 1                  !    9
```

Часто логические выражения с целочисленными операндами применяются для маскирования тех или иных разрядов.

Пример маскирования старшего байта:

```
integer(2) :: mask = #00ff        ! Маска mask и число k заданы в
integer(2) :: k = #5577           ! шестнадцатеричной системе счисления
write(*, '(z)') mask .and. k      !    77 (в шестнадцатеричной системе счисления)
```

Операции отношения и логические операции выполняются слева направо, т. е., если две последовательные операции имеют равный приоритет, первоначально выполняется левая операция.

Пример. Вычислить результат логического выражения

x / a == 1 .or. b / (a + b) < 1 .and. .not. b == a .or. x /= 6

при $x = 6.0$, $a = 2.0$ и $b = 3.0$.

Вычислив результат арифметических операций и операций отношения, получим:

.false. .or. .true. .and. .not. .false. .or. .false.

Далее выполняем пошагово логические операции с учетом их приоритета. После выполнения .not. .false.:

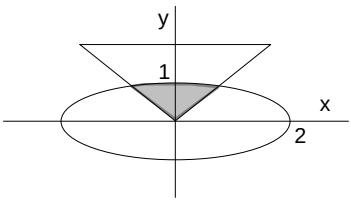
.false. .or. .true. .and. .true. .or. .false.

После выполнения .true. .and. .true.:

.false. .or. .true. .or. .false.

Окончательный результат: .TRUE.

Пример. Записать условие попадания точки в область, которая является пересечением эллипса и треугольника, образованного графиками функций $y = |x|$ и $y = 2$ (рис. 5.1).




```
if(x**2/4 + y**2 < 1.0 .and. y > abs(x)) then
  write(*, *)  'Inside'
else
  write(*, *)  'Outside'
end if
```

Рис. 5.1. Исследуемая область

Логической переменной можно присвоить значение целочисленного выражения, которое интерпретируется как *истина*, если отлично от нуля, и как *ложь*, если равно нулю. С другой стороны, логические величины можно использовать в арифметических выражениях. В этом случае .TRUE. интерпретируется как единица, а .FALSE. - как нуль. И как следствие этого свойства, результат логического выражения можно присвоить числовой переменной. Однако если логическая переменная, например *g*1, получила свое значения, например 11, в результате вычисления целочисленного выражения, то при последующем использовании *g*1 в арифметическом выражении ее значение будет равно 11, а не единице. Например:

```
integer :: k = 22, m = 0
logical g1, g2
g1 = k / 2; g2 = m * k
print *, g1, g2                          ! T  F
print *, 3*g1, 3*(.not.g2), 3**g2        !   33     -3       1
k = .not. g1 .or. .not. g2
print *, k                               !   -1
```

Замечание. Свойства CVF и FPS, позволяющие смешивать логические и целочисленные данные, являются расширением по отношению к стандарту Фортран 90.

5.3. Задаваемые операции

Действие встроенных операций (одноместных и двуместных) может быть распространено на производные типы данных, для которых не определено ни одной встроенной операции. Механизм расширения области действия операции называется *перегрузкой операции*. Помимо этого могут быть заданы и дополнительные операции.

Механизм перегрузки и задания двуместной операции *x op y*:

- составить функцию *fop* с двумя обязательными параметрами *x* и *y*, имеющими вид связи IN, которая будет вызываться для реализации задаваемой операции *op* с операндами *x* и *y* и будет возвращать результат операции;
- при помощи оператора INTERFACE OPERATOR(*op*) связать функцию *fop* с операцией *op*.

Тогда результатом операции *x op y* будет возвращаемое функцией *fop*(*x*, *y*) значение, т. е. следующие операторы эквивалентны:

```
z = x op y
z = fop(x, y)
```

Аналогично реализуется механизм задания и перегрузки одноместной операции (разд. 8.12.2).

Пример. Задать операцию вычисления процента *x* от *y*.

```
interface operator(.c.)                 ! Связываем операцию .c.
 real(4) function percent(x, y)         ! с функцией percent
  real(4), intent(in) :: x, y
 end function percent
end interface
print '(1x,f5.1)', 5.0 .c. 10.0         ! 50.0
print '(1x,f5.1)', percent(5.0, 10.0)   ! 50.0
end

real(4) function percent(x, y)      ! Эта функция вызывается при
 real(4), intent(in) :: x, y        ! выполнении операции .c.
 percent = x / y * 100.0
end function percent
```

Замечание. Реализующая операцию функция может быть модульной процедурой (разд. 8.12.2).

Задаваемая операция должна всегда обрамляться точками. Типы операндов задаваемой операции должны строго соответствовать типам параметров вызываемой при выполнении операции функции. Так, в нашем примере попытка выполнить операцию 5 .c. 10 приведет к ошибке, поскольку типы операндов отличны от REAL(4).

При перегрузке операций отношения, для обозначения которых существует две формы, перегрузка распространяется на обе формы операции. Например, если перегружена операция >=, то таким же образом будет перегружена и операция .GE..

Более подробно механизмы задания и перегрузки операций изложены в разд. 8.12.

5.4. Приоритет выполнения операций

Когда арифметические, символьные, логические операции и операции отношения присутствуют в одном выражении (такая смесь может быть, например, в логическом выражении), приоритет выполнения операций таков (дан в порядке убывания):

1) любая заданная или перегруженная одноместная операция;
2) арифметические операции;
3) символьная операция конкатенации;
4) операции отношения;
5) логические операции;
6) любая заданная или перегруженная двуместная операция.

В табл. 5.2 встроенные операции Фортрана расположены в порядке убывания приоритета.

Таблица 5.2. Приоритет выполнения встроенных операций

**	*	+	//	.EQ., ==	.NOT.	.AND.	.OR.	.XOR.
	/	-		.NE., /=				.EQV.
				.LT., <				.NEQV.
				.LE., <=				
				.GT., >				
				.GE., >=				

Замечание. Каждая ячейка таблицы содержит операции с равным приоритетом.

5.5. Константные выражения

В операторах объявления Фортрана могут появляться выражения (например, при задании значений именованных констант), но такие выражения должны быть *инициализирующими* и *константными*, например:

```
integer, parameter :: n = 10, m = n / 2
real a(m, n), b(2 * n)                ! n / 2 и 2 * n - примеры константных выражений
```

В общем случае константное выражение - это выражение, в котором все операции встроенные, а каждый простой элемент - это:

- константное выражение, заключенное в скобки;
- константа или подобъект константы, в котором каждый индекс сечения или граница подстроки является константным выражением;
- конструктор массива, в выражениях которого (включая границы и шаги) каждый простой член является константным выражением или переменной неявного цикла;

- конструктор структуры, компоненты которого являются константными выражениями;
- обращение к встроенной элементной или преобразовывающей функции, все параметры в котором являются константными выражениями;
- обращение к встроенной справочной функции (кроме функций PRESENT, ASSOCIATED или ALLOCATED), в котором каждый параметр - это либо константное выражение, либо переменная, о которой выдается справка. Причем границы переменной, о которой выдается справка, не должны быть подразумеваемыми (случай массива или строки, перенимающей размер) или заданы с помощью оператора ALLOCATE или путем прикрепления ссылки.

Именованным константам (объектам с атрибутом PARAMETER) могут быть присвоены значения только *инициализирующих* константных выражений. Значения таких выражений вычисляются при компиляции, и поэтому на них накладываются дополнительные ограничения:

- допускается возведение в степень лишь с целым показателем;
- аргументы и результаты встроенных элементных функций должны быть целого или текстового типа;
- из преобразовывающих функций допускаются только REPEAT, RESHAPE, SELECTED_INT_KIND, SELECTED_REAL_KIND, TRANSFER и TRIM.

Каждый элемент инициализирующего выражения должен быть определен в предшествующем операторе объявления или левее в том же самом операторе объявления.

Пример:

```
character(*), parameter :: st(3) = (/ 'Январь', 'Февраль', 'Март' /)
integer, parameter :: n = len_trim(st(2))
```

5.6. Описательные выражения

При задании в процедурах параметров разновидностей типов, границ массивов и текстовых длин объектов данных, а также при задании результатов функций могут наряду с константными использоваться скалярные, неконстантные выражения. Такие выражения называются *описательными* и содержат ряд ограничений:

- они могут зависеть только от тех значений, которые определены при входе в процедуру;
- типы и параметры типов переменных описательного выражения должны объявляться ранее их использования в выражении, за исключением случаев, когда тип переменной определяется в соответствии с правилами умолчания о типах данных.

В состав описательных выражений могут входить конструкторы массивов, производных типов и обращения к встроенным функциям. Но последние ограничены:

- элементными функциями с параметрами и результатом целого или текстового типа;
- функциями REPEAT, RESHAPE, TRANSFER и TRIM с параметрами целого или текстового типа;
- справочными функциями, кроме функций PRESENT, ASSOCIATED и ALLOCATED, при условии, что величина, информация о которой выдается, не зависит от выделения памяти и прикрепления ссылки.

В обращении к справочной функции исследуемый объект может быть доступен посредством *use*-ассоциирования или ассоциирования через носитель или может быть объявлен в том же программном компоненте, но обязательно до его использования в справочной функции. На элемент массива, объявляемого в программном компоненте, можно ссылаться только после описания его границ.

Пример:

```
function fun(x, y, lob)
 real x                          ! Прежде объявляем x
 real(kind(x)) y, fun            ! Теперь можно употребить x в описательном
 integer lob(2)                  ! выражении
 real, dimension( lob(1) : max(lob(2), 10) ) :: z
 real wz(lob(2) : size(z))
! Параметр разновидности типа переменной y и результирующей переменной
! fun, границы массивов z и wz задаются описательными выражениями.
```

5.7. Присваивание

Присваивание является оператором, в результате выполнения которого переменная получает значение расположенного в правой части оператора присваивания выражения. Переменная, получающая значение выражения, может быть как скаляром, так и массивом. В результате присваивания значения могут получать как объекты, так и их подобъекты, например элементы массивов, подстроки, компоненты переменных производных типов, сечения массивов.

Синтаксис оператора:

var = expr

где *var* - имя переменной; *expr* - выражение.

В соответствии со стандартом в случае *встроенного* присваивания типы переменной *var* и выражения *expr* должны соответствовать друг другу:

- результат арифметического выражения может быть присвоен числовой переменной. Если переменная и выражение имеют разные числовые

типы, то тип результата выражения приводится к типу переменной, которой присваивается результат, например:

```
complex :: z = (-2.0, 3.0)
integer k
k = z * z
print *, k                    !     -5
```

- результат логического выражения может быть присвоен логической переменной;
- результат символьного выражения может быть присвоен только символьной переменной;
- переменной производного типа можно присвоить значение выражения данного типа, например:

```
type pair
  real x, y
end type pair
type(pair) :: p1, p2 = pair(1.0, 2.0)
p1 = p2
p2 = pair(-2.0, -5.0)
```

Правда, как мы уже видели, CVF и FPS имеют расширения по отношению к стандарту языка: логической переменной можно присвоить результат целочисленного выражения и, наоборот, числовой переменной можно присвоить результат логического выражения.

Можно, однако, выполнить *перегрузку* присваивания. Например, можно задать оператор присваивания, в котором *var* имеет числовой, а *expr* - символьный тип. Или задать присваивание, при котором переменной производного типа присваивается результат выражения иного типа. Перегрузка присваивания выполняется так:

- составить подпрограмму *sub* с двумя обязательными параметрами *x* и *y*, причем параметр *x* должен иметь вид связи OUT, параметр *y* - IN. В результате работы подпрограммы определяется значение *x*;
- используя оператор INTERFACE ASSIGNMENT(=), связать подпрограмму *sub* с оператором присваивания. Тогда подпрограмма будет вызываться каждый раз, когда *var* имеет такой же тип, как и *x*, а тип *expr* совпадает с типом параметра *y*, т. е. присваивание *x = y* эквивалентно вызову *call sub*(*x*, *y*).

Пример. Присвоить целочисленной переменной сумму кодов символов строки.

```
interface assignment(=)
 subroutine charti(n, ch)
  integer, intent(out) :: n
  character(*), intent(in) :: ch
```

```
 end subroutine charti
end interface
integer k, m
character(80) :: st = 'String to count'
k = st(:10)                          ! Выполняется заданное присваивание
call charti(m, st(:10))              ! Этот вызов эквивалентен оператору m = st(:10)
print *, k, m                        !       890       890
end

subroutine charti(n, ch)
 integer, intent (out) :: n
 character(*), intent (in) :: ch
 integer i
 n = 0
 do i = 1, len_trim(ch)
  n = n + ichar(ch(i:i))
 end do
end subroutine charti
```

Замечание. Реализующая присваивание подпрограмма может быть модульной процедурой (разд. 8.7).

Более подробно вопросы перегрузки присваивания рассмотрены в разд. 8.12.2.

6. Встроенные процедуры

6.1. Виды встроенных процедур

Встроенные процедуры разделяются на 4 вида:

1) *элементные процедуры*. Параметрами таких процедур могут быть как скаляры, так и согласованные массивы. Когда параметрами являются массивы, каждый элемент результирующего массива равен результату применения процедуры к соответствующим элементам массивов-параметров. Среди элементных процедур есть одна подпрограмма - MVBITS. Остальные являются функциями. Результирующий массив должен быть согласован с массивами-параметрами;
2) *справочные функции* выдают информацию о свойствах параметров функций. Результат справочной функции не зависит от значения параметра, который, в частности, может быть и неопределенным;
3) *преобразовывающие функции*;
4) *неэлементные подпрограммы*.

Помимо встроенных CVF и FPS имеют большое число дополнительных процедур, перечень которых приведен в прил. 3.

6.2. Обращение с ключевыми словами

Поскольку встроенные процедуры обладают явно заданным интерфейсом, их вызов может быть выполнен с ключевыми словами. Например:

```
pi = asin(x = 1.0)                    ! x - ключевое слово
y = sin(x = pi)
```

В качестве ключевых слов используются имена формальных параметров. Эти имена приводятся при описании каждой встроенной процедуры. Необходимость в таких вызовах возникает лишь при работе с процедурами, которые имеют необязательные параметры, и в том случае, когда изменяется естественный порядок следования параметров. Иногда применение ключевых слов делает вызов более наглядным.

Пример. Функция MAXVAL(*array* [, *dim*] [, *mask*]) поиска максимальных значений в массиве имеет два необязательных параметра *dim* и *mask*.

```
integer array(2, 3)
array = reshape((/1, 4, 5, 2, 3, 6/), (/2, 3/))
! Массив array:    1  5  3
!                  4  2  6
! Естественный порядок параметров. Вызов без ключевых слов
print 1, maxval(array, 1, array < 4)                    !    1     2     3
print 1, maxval(array, 1)                               !    4     5     6
! Вызов с ключевыми словами; однако поскольку порядок параметров
```

```
! не нарушен, применение ключевых слов необязательно
print 1, maxval(array, dim = 1, mask = array < 4)      !   1   2   3
print 1, maxval(array, dim = 1)                        !   4   5   6
! Применение ключевых слов обязательно
print 1, maxval(array, mask = array < 4, dim = 1)      !   1   2   3
print 1, maxval(array, mask = array < 4)               !   3
1 format(3i3)
end
```

6.3. Родовые и специфические имена

Имена многих встроенных процедур являются родовыми. Например, родовым является имя функции MAXVAL из только что приведенного примера. На практике это означает, что тип результата такой функции зависит от типа обязательного параметра. Так, функция может принимать в качестве параметра *array* массив любого целого или вещественного типа. Тип и значение параметра разновидности типа результата функции MAX-VAL такие же, как у параметра *array*. Родовые имена встроенных функций не могут быть использованы в качестве фактических параметров процедур (разд. 8.18.2).

Ряд встроенных функций имеют специфические имена, и их вызов может быть выполнен как по родовому, так и по специфическому имени. Вызов с родовым именем и параметром типа *type* эквивалентен вызову со специфическим именем с тем же типом параметра.

Пример:

```
real(4) :: x = 3.0
real(8) :: y = 3.0                  ! LOG  - родовое имя
complex(4) :: z = (3.0, 4.0)        ! CLOG - специфическое имя
print *, log(x)                     !      1.098612
print *, log(y)                     !      1.098612288668110
print *, log(z)                     !      (1.609438, 9.272952E-01)
print *, clog(z)                    !      (1.609438, 9.272952E-01)
```

Замечание. Вызов функции LOG с целочисленным параметром недопустим, поскольку в противном случае не было бы ясности, в какой допустимый тип (вещественный или комплексный) следует автоматически преобразовать целочисленный аргумент. То же справедливо и для других математических функций.

Специфические имена функций могут быть использованы в качестве параметров процедур (см. табл. 8.3 в разд. 8.18.2) за исключением имен, указанных в табл. 8.4. Специфические имена применяют также, когда необходимо сделать очевидным для программиста используемый в расчетах тип данных.

В последующем описании встроенных процедур будут упоминаться только их родовые имена. Для справок относительно их специфических имен мы вновь отсылаем к табл. 8.3 и 8.4 в разд. 8.18.2.

6.4. Возвращаемое функцией значение

Ряд функций, например RANGE или INDEX, возвращают значение стандартного целого типа (INTEGER). По умолчанию этот тип эквивалентен типу INTEGER(4). Однако если применена опция компилятора /4I2 или директива $INTEGER:2, то возвращаемый функцией результат имеет тип INTEGER(2). При этом также меняется устанавливаемый по умолчанию тип логических данных, т. е. функция стандартного логического типа, например ASSOCIATED, будет возвращать результат типа LOGICAL(2), а не LOGICAL(4).

То же справедливо и для функций, возвращающих значение стандартного вещественного типа, который по умолчанию эквивалентен REAL(4). Однако если пользователем задана опция компилятора /4R8 или директива $REAL:8, то возвращаемый функцией результат имеет тип REAL(8).

6.5. Элементные функции преобразования типов данных

В выражениях Фортрана можно использовать операнды разных типов. При вычислениях типы данных будут преобразовываться в соответствии с рангом типов операндов. Однако часто требуется явное преобразование типов, например чтобы избежать целочисленного деления или правильно обратиться к функции. Для подобных целей используются функции преобразования типов данных. Например:

```
integer :: a = 2, b = 3
print *, sin(float(a + b))          ! -9.589243E-01
```

AIMAG(*z*) - возвращает мнимую часть комплексного аргумента *z*. Результат имеет вещественный тип с параметром разновидности, таким же, как и у *z*.

INT(*a* [, *kind*]) - преобразовывает параметр *a* в целый тип с параметром разновидности *kind* путем отсечения значения *a* в сторону нуля. Тип параметра *a* - целый, вещественный или комплексный. Если параметр *a* комплексного типа, то действительная часть преобразовывается в целый тип путем отсечения в сторону нуля. Если параметр *kind* отсутствует, то результат имеет стандартный целый тип. Тип *kind* - INTEGER.

Аналогичные преобразования, но с фиксированным типом результата выполняются следующими функциями:

Функция	*Типы параметра*	*Тип результата*
INT1(*a*)	Целый, вещественный или комплексный	INTEGER(1)

INT2(*a*)	" " " "	INTEGER(2)
INT4(*a*)	Целый, вещественный или комплексный	INTEGER(4)
HFIX(*a*)	" " " "	INTEGER(2)
JFIX(*a*)	" " " "	INTEGER(4)

IZEXT(*a*), JZEXT(*a*) и ZEXT(*a*) - преобразовывают логические и целые значения в целый тип с большим значением параметра разновидности. Преобразование выполняется путем добавления нулей в свежие биты результата:

Функции	*Типы параметра*	*Тип результата*
IZEXT(*a*)	LOGICAL(1), LOGICAL(2), BYTE, INTEGER(1), INTEGER(2)	INTEGER(2)
JZEXT(*a*) и ZEXT(*a*)	LOGICAL(1), LOGICAL(2), LOGICAL(4), BYTE, INTEGER(1), INTEGER(2), INTEGER(4)	INTEGER(4)

REAL(*a* [, *kind*]) - преобразовывает параметр *a* в вещественный тип с параметром разновидности *kind.* Тип параметра *a* - целый, вещественный или комплексный. Если параметр *kind* отсутствует, то результат имеет стандартный вещественный тип. Если параметр *a* комплексного типа, то результат вещественный с параметром разновидности типа *kind*, если *kind* задан, и с тем же параметром разновидности типа, что и *a*, если *kind* опущен.

DBLE(*a*) и DFLOAT(*a*) - преобразовывают целый, вещественный или комплексный параметр *a* в вещественный тип REAL(8).

CMPLX(*x* [, *y*] [, *kind*]) - преобразовывает целые, вещественные или комплексные параметры в комплексный тип с параметром разновидности *kind*. Если параметр *kind* опущен, то результат COMPLEX(4). Если *y* задан, то *x* - вещественная часть комплексного результата. Параметр *y* не может быть задан, если *x* комплексного типа. Если *y* задан, то он является мнимой частью комплексного результата. Если *x* и *y* являются массивами, то они должны быть согласованными.

Пример:

```
complex z1, z2
complex(8) z3
z1 = cmplx(3)          ! Возвращает 3.0 + 0.0i
z2 = cmplx(3, 4)       ! Возвращает 3.0 + 4.0i
z3 = cmplx(3, 4, 8)    ! Возвращает число типа COMPLEX(8) 3.0d0 + 4.0d0i
```

DCMPLX(*x* [, *y*]) - выполняет те же преобразования, что и функция CMPLX, но тип результата всегда COMPLEX(8). Тип параметров *x* и *y* - целый, вещественный или комплексный.

LOGICAL(*L* [, *kind*]) - преобразовывает логическую величину из одной разновидности в другую. Результат имеет такое же значение, что и *L* и параметр разновидности *kind*. Если *kind* отсутствует, то тип результата LOGICAL.

TRANSFER(*source*, *mold* [, *size*]) - переводит данные *source* в другой тип без изменения физического представления данных, т. е. значения отдельных битов результата и *source* совпадают. Тип и параметры типа результата такие же, как у *mold*.

Пусть физическое представление *source* есть последовательность *n* бит $b_1 b_2 \ldots b_n$, а представление *mold* занимает *m* бит, тогда результат:

при $n = m$ равен $b_1 b_2 \ldots b_n$;

при $n < m$ - $b_1 b_2 \ldots b_n s_1 s_2 \ldots s_{m-n}$, где биты s_i не определены;

при $n > m$ - $b_1 b_2 \ldots b_m$.

Если *mold* скаляр и параметр *size* опущен, то результат является скаляром. Если *mold* массив и *size* опущен, то результатом является одномерный массив, размер которого достаточен для размещения в нем *source*. Если параметр *size* задан, то результатом является одномерный массив размером *size*.

Ниже даны элементные функции преобразования символа в его целочисленное представление и функции обратного преобразования, возвращающие символ по его коду. Описание функций приведено в разд. 3.8.8.

Функция	*Тип параметра*	*Тип результата*
ICHAR(*c*)	CHARACTER(1)	INTEGER(4)
IACHAR(*c*)	"	"
CHAR(*i* [, *kind*])	Целый	CHARACTER(1)
ACHAR(*i*)	"	"

6.6. Элементные числовые функции

ABS(*a*) - абсолютная величина целого, вещественного или комплексного аргумента. Если *a* целого типа, то и результат целого типа, в остальных случаях результат будет вещественным. Для комплексного аргумента $a = x + y\,i$: ABS(*a*) = SQRT(*x***2 + *y***2).

Пример:

```
complex(4) :: z =(3.0, 4.0)
write(*, *) abs(z)                ! 5.0        (тип результата - REAL(4))
```

AINT(*a* [, *kind*]) - обрезает вещественную величину *a* в сторону нуля до целого числа и выдает результат в виде вещественной величины, разновидность типа которой совпадает со значением аргумента *kind*, если

он задан, или - в противном случае - со стандартной разновидностью вещественного типа.

ANINT(*a* [, *kind*]) - возвращает в виде вещественной величины целое число, ближайшее к значению вещественного аргумента *a* (выполняет округление *a*). Разновидность типа результата совпадает со значением аргумента *kind*, если он задан, или - в противном случае - со стандартной разновидностью вещественного типа.

NINT(*a* [, *kind*]) - возвращает целое число, ближайшее к значению вещественного аргумента *a* (выполняет округление *a*). Разновидность типа результата совпадает со значением аргумента *kind*, если он задан, или - в противном случае - со стандартной разновидностью целого типа.

Пример:

```
real :: a(3) = (/ 2.8, -2.8, 1.3 /)
write(*, *) anint(a)                ! 3.000000  -3.000000  1.000000
write(*, *) nint(2.8), nint(2.2)    !  3   2
write(*, *) nint(a(2)), nint(-2.2)  !  -3  -2
write(*, *) aint(2.6), aint(-2.6)   ! 2.000000  -2.000000
```

CEILING(*a* [, *kind*]) - возвращает наименьшее целое, большее или равное значению вещественного аргумента *a*. Разновидность типа результата совпадает со значением аргумента *kind*, если он задан, или - в противном случае - со стандартной разновидностью целого типа. Необязательный параметр добавлен стандартом 1995 г.

CONJG(*z*) - возвращает комплексное число, сопряженное со значением комплексного аргумента *z*.

Пример:

```
print *, conjg((3.0, 5.6))          ! (3.000000, -5.600000)
```

DIM(*x, y*) - возвращает $x - y$, если $x > y$, и 0, если $x \le y$. Аргументы *x* и *y* должны быть оба целого или вещественного типа.

Пример:

```
print *, dim(6, 4), dim(4.0, 6.0)   !  2   0.000000E+00
```

DPROD(*x*, *y*) - возвращает произведение двойной точности - REAL(8). Аргументы *x* и *y* должны быть стандартного вещественного типа.

Пример:

```
real :: a = 3.72382, b = 2.39265
write(*, *) a * b, dprod(a, b)
! Результат:        8.9097980        8.90979744044290
```

FLOOR(*a* [, *kind*]) - возвращает наибольшее целое, меньшее или равное значению вещественного аргумента *a*. Разновидность типа результата совпадает со значением аргумента *kind*, если он задан, или - в противном

случае - со стандартной разновидностью целого типа. Необязательный параметр добавлен стандартом 1995 г.

Пример для CEILING и FLOOR:

```
integer i, iarray(2)
i = ceiling(8.01)                          ! Возвращает        9
i = ceiling(-8.01)                         ! Возвращает       -8
iarray = ceiling((/8.01, -5.6/))           ! Возвращает  (9, -5)
i = floor(8.01)                            ! Возвращает        8
i = floor(-8.01)                           ! Возвращает       -9
iarray = floor((/ 8.01, -5.6 /), kind = 2) ! Возвращает (8, -6) типа INTEGER(2)
```

MOD(*a*, *p*) - возвращает остаток от деления *a* на *p*, т. е. MOD(*a, p*) = *a* - - INT*(a*/*p*)**p*. Параметры *a* и *p* должны быть либо оба целыми, либо оба вещественными. Если *p* = 0, то результат не определен.

Пример:

```
write(*, *) mod(5, 3), mod(5.3, 3.0)          !      2      2.300000
```

MODULO(*a, p*) - возвращает *a* по модулю *p*. Параметры *a* и *p* должны быть либо оба целыми, либо оба вещественными. Результат *r* таков, что *a* = *q* * *p* + *r*, где *q* - целое число; |*r*| < *p*, и *r* имеет тот же знак, что и *p*. Если *p* = 0, то результат не определен. Для вещественных *a* и *p* MODULO(*a, p*) = *a* - FLOOR(*a* / *p*) * *p* .

Пример:

```
print *, modulo(8, 5)          !          3          (q =  1)
print *, modulo(-8, 5)         !          2          (q = -2)
print *, modulo(8, -5)         !         -2          (q = -2)
print *, modulo(7.285, 2.35)   !  2.350001E-01       (q =  3)
print *, modulo(7.285, -2.35)  !       -2.115        (q = -4)
```

SIGN(*a, b*) - возвращает абсолютную величину *a*, умноженную на +1, если *b* ≥ 0, и -1, если *b* < 0. Параметры *a* и *b* должны быть либо оба целыми, либо оба вещественными.

Пример. Функция SIGN вернет 1.0, если второй ее аргумент - *ya* - больше нуля или равен ему, и -1.0 - в противном случае.

```
result = sign(1.0, ya)
```

6.7. Вычисление максимума и минимума

Функции нахождения максимума и минимума являются элементными и применимы к числовым данным вещественного и целого типа. Имена MAX и MIN являются родовыми.

AMAX0(*a*1, *a*2, [, *a*3, ...]) - возвращает максимум из двух или более значений стандартного целого типа. Результат имеет стандартный вещественный тип.

MAX(*a*1, *a*2, [, *a*3, ...]) - возвращает максимум из двух или более целых или вещественных значений. Тип и разновидность типа результата совпадают с типом параметров.

MAX1(*a*1, *a*2, [, *a*3, ...]) - возвращает максимум из двух или более значений стандартного вещественного типа. Результат имеет стандартный целый тип.

AMIN0(*a*1, *a*2, [, *a*3, ...]) - возвращает минимум из двух или более значений стандартного целого типа. Результат имеет стандартный вещественный тип.

MIN(*a*1, *a*2, [, *a*3, ...]) - возвращает минимум из двух или более целых или вещественных значений. Тип и разновидность типа результата совпадают с типом параметров.

MIN1(*a*1, *a*2, [, *a*3, ...]) - возвращает минимум из двух или более значений стандартного вещественного типа. Результат имеет стандартный целый тип.

Во всех случаях параметры функций - скалярные выражения.

Пример:

```
write(*, *) max1(5.2, 3.6, 9.7)          !      9
write(*, *) amin0(5, -3, 9)              !     -3.0
```

6.8. Математические элементные функции

Фортран содержит математические функции вычисления корня, логарифмов, экспоненты и тригонометрических функций. Тип и параметр разновидности типа результата такие же, как у первого аргумента. В разделе приведены формы вызова функций с родовыми именами. Специфические имена функций даны в разд. 8.18.2.

Когда параметрами логарифмических и тригонометрических функций являются комплексные числа, то функции возвращают комплексное число, аргумент ϑ которого равен главному значению аргумента комплексного числа в радианах ($-\pi < \vartheta \le \pi$).

6.8.1. Экспоненциальная, логарифмическая функции и квадратный корень

EXP(*x*) - возвращает e^x = *e****x* для вещественного или комплексного *x*. В случае комплексного *x* = (*a, b*) результат равен *e****a**(cos(*b*) + *i* sin(*b*)).

LOG(*x*) - возвращает значение натурального логарифма для вещественного или комплексного *x*. В случае вещественного аргумента значение *x* должно быть больше нуля. В случае комплексного аргумента *x* не должен быть нулем. Если *x* комплексного типа, то действительный компонент результата равен натуральному логарифму модуля *x*, мнимый компонент - главному значению аргумента *x* в радианах, т. е. если *x* = (*a, b*), то LOG(*x*) = = (LOG(SQRT(*a***2 + *b***2)), ATAN2(*a, b*)).

LOG10(*x*) - возвращает десятичный логарифм вещественного аргумента. Значение *x* должно быть больше нуля.

SQRT(*x*) - возвращает квадратный корень для вещественного или комплексного аргумента *x*. В случае вещественного аргумента значение *x* должно быть больше нуля. В случае комплексного *x* функция возвращает число, модуль которого равен корню квадратному из модуля *x* и угол которого равен половине угла *x*. Так, если *x* = *(a, b)*, то SQRT(*x*) = SQRT(*a***2 + *b***2) * * $e^{-j * 0.5 * \text{atan}(a / b)}$.

Извлечь корень можно также, применив операцию возведения в степень: SQRT(*x*) = *x***0.5. Однако применять следует функцию SQRT, поскольку SQRT(*x*) выполняется быстрее, чем *x***0.5.

6.8.2. Тригонометрические функции

В Фортране существуют как тригонометрические функции, в которых аргумент должен быть задан в радианах, например SIN(*x*), так и функции, аргумент которых задается в градусах, например SIND(*x*).

Синус и арксинус

SIN(*x*) - возвращает синус вещественного или комплексного аргумента *x*, который интерпретируется как значение в радианах.

SIND(*x*) - возвращает синус вещественного или комплексного аргумента *x*, который интерпретируется как значение в градусах.

ASIN(*x*) - возвращает арксинус вещественного аргумента *x* ($|x| \le 1$), выраженный в радианах в интервале $-\pi/2 \le \text{ASIN}(x) \le \pi/2$.

ASIND(*x*) - возвращает арксинус вещественного аргумента *x* ($|x| \le 1$), выраженный в градусах в интервале $-90 \le \text{ASIN}(x) \le 90$.

Косинус и арккосинус

COS(*x*) - возвращает косинус вещественного или комплексного аргумента *x*, который интерпретируется как значение в радианах.

COSD(*x*) - возвращает косинус вещественного или комплексного аргумента *x*, который интерпретируется как значение в градусах.

ACOS(*x*) - возвращает арккосинус вещественного аргумента *x* ($|x| \le 1$), выраженный в радианах в интервале $0 \le \text{ACOS}(x) \le \pi$.

ACOSD(*x*) - возвращает арккосинус вещественного аргумента *x* ($|x| \le 1$), выраженный в градусах в интервале $0 \le \text{ACOS}(x) \le 180$.

Тангенс, котангенс и арктангенс

TAN(*x*) - возвращает тангенс вещественного аргумента *x*, который интерпретируется как значение в радианах.

TAND(*x*) - возвращает тангенс вещественного аргумента *x*, который интерпретируется как значение в градусах.

COTAN(*x*) - возвращает котангенс вещественного аргумента *x* ($x \ne 0$), который интерпретируется как значение в радианах.

ATAN(*x*) - возвращает арктангенс вещественного аргумента *x*, выраженный в радианах в интервале $-\pi/2 < ATAN(x) < \pi/2$.

ATAND(*x*) - возвращает арктангенс вещественного аргумента *x*, выраженный в градусах в интервале $-90 < ATAN(x) < 90$.

ATAN2(*y, x*) - возвращает арктангенс (*y*/*x*), выраженный в радианах в интервале $-\pi \le ATAN2(y, x) \le \pi$. Аргументы *y* и *x* должны быть вещественного типа с одинаковым значением параметра разновидности и не могут одновременно равняться нулю.

ATAN2D(*y, x*) - возвращает арктангенс (*y*/*x*), выраженный в градусах в интервале $-180 \le ATAN2D(y, x) \le 180$. Аргументы *y* и *x* должны быть вещественного типа с одинаковым значением параметра разновидности и не могут одновременно равняться нулю.

Диапазон результатов функций ATAN2 и ATAN2D:

Аргументы	*Результат (в рад*)	*Результат (в град.*)
$y > 0$	Результат > 0	Результат > 0
$y = 0$ и $x > 0$	Результат = 0	Результат = 0
$y = 0$ и $x < 0$	Результат = π	Результат = 180
$y < 0$	Результат < 0	Результат < 0
$x = 0$ и $y > 0$	Результат = $\pi/2$	Результат = 90
$x = 0$ и $y < 0$	Результат = $-\pi/2$	Результат = -90

Гиперболические тригонометрические функции

SINH(*x*) - гиперболический синус для выраженного в радианах вещественного аргумента *x*.

COSH(*x*) - гиперболический косинус для выраженного в радианах вещественного аргумента *x*.

TANH(*x*) - гиперболический тангенс для выраженного в радианах вещественного аргумента *x*.

6.9. Функции для массивов

Встроенные функции для работы с массивами позволяют выполнять вычисления в массивах, получать справочные данные о массиве и преобразовывать массивы. Встроенные функции обработки массивов рассмотрены в разд. 4.12. Помимо встроенных CVF и FPS содержат дополнительные подпрограмму SORTQQ сортировки одномерного массива и целочисленную функцию BSEARCHQQ бинарного поиска в отсортированном массиве. Для их вызова необходимо сослаться на модуль MSFLIB.

CALL SORTQQ(*adrarray*, *count*, *size*) - сортирует одномерный массив, адрес которого равен *adrarray*. Для вычисления адреса применяется

функция LOC. Сортируемый массив не должен быть производного типа. Параметр *count* имеет вид связи INOUT и при вызове равен числу элементов массива, подлежащих сортировке, а на выходе - числу реально отсортированных элементов. Тип параметров *adrarray*, *count* - стандартный целый.

Параметр *size* является положительной константой стандартного целого типа (*size* < 32,767), задающей тип и разновидность типа сортируемого массива. В файле msflib.f90 определены следующие константы:

Константа	*Типы массива*
SRT$INTEGER1	INTEGER(1)
SRT$INTEGER2	INTEGER(2) или эквивалентный
SRT$INTEGER4	INTEGER(4) или эквивалентный
SRT$REAL4	REAL(4) или эквивалентный
SRT$REAL8	REAL(8) или эквивалентный

Если величина *size* не является именованной константой приведенной таблицы и меньше чем 32'767, то предполагается, что задан символьный массив, длина элемента которого равна *size*.

Чтобы убедиться в том, что сортировка выполнена успешно, следует сравнить значения параметра *count* до и после сортировки. При положительном результате они совпадают.

Предупреждение. Адрес сортируемого массива должен быть вычислен функцией LOC. Значения параметров *count* и *size* должны точно описывать характеристики массива. Если же подпрограмма SORTQQ получила неверные параметры, то будет выполнена попытка сортировки некоторой области памяти. Если память принадлежит текущему процессу, то сортировка будет выполнена, иначе операционная система выполнит функции защиты памяти и остановит вычисления.

BSEARCHQQ(*adrkey*, *adrarray*, *length*, *size*) - выполняет бинарный поиск значения, которое содержится в переменной, расположенной по адресу *adrkey*. Поиск выполняется в отсортированном одномерном массиве, первый элемент которого имеет адрес *adrarray*. Функция возвращает индекс искомого элемента или 0, если элемент не найден. Тип результата и параметров *adrkey*, *adrarray*, *length* и *size* - стандартный целый. Элементы массива не могут быть производного типа. Параметр *length* равен числу элементов массива. Смысл параметра *size* пояснен при рассмотрении подпрограммы SORTQQ.

До выполнения поиска массив должен быть отсортирован по возрастанию значений его элементов.

Предупреждение. Адреса сортируемого массива и элемента должны быть вычислены функцией LOC. Значения параметров *count* и *size* должны точно описывать характеристики массива. К тому же искомый элемент должен иметь тот же тип и разновидность типа, что и массив, в котором выполняется поиск. Эти характеристики задаются параметром *size*. Если же функция BSEARCHQQ получила неверные параметры, то, если память принадлежит текущему процессу, будет выполнена попытка поиска в некоторой области памяти, иначе операционная система выполнит функции защиты памяти и остановит вычисления.

Пример. Найти в массиве все равные заданному значению элементы.

Для решения предварительно отсортируем массив, а далее выполним поиск, учитывая, что равные элементы отсортированного массива следуют подряд.

```
use msflib
integer(4) a(20000), n, n2, ada, i, k
integer(4) :: ke = 234                    ! Вычислим, сколько раз в массиве a
real(4) rv                                ! встречается число ke
n = size(a); n2 = n                       ! Запомним размер массива a
do i = 1, n                               ! Заполним массив a случайным образом
 call random(rv)
 a(i) = int(rv * 1000.0)
end do
ada = loc(a)                              ! Адрес массива a
call sortqq(ada, n, SRT$INTEGER4)         ! Сортировка массива a
if(n .ne. n2 ) stop 'Sorting error'
k = bsearchqq(loc(ke), ada, n, SRT$INTEGER4)
if(k == 0) then
 print *, 'Элемент ke = ', ke, ' не найден'
 stop
end if
i = k                                     ! Поиск всех равных ke элементов
do while(a(i) == ke .and. i <= n)
 print *, 'Элемент с индексом i = ', i, ' равен ke; ke = ', ke
 i = i + 1
end do
print *, 'Всего найдено элементов: ', i - k      !     21
end
```

На самом деле результат неверен. Действительно, вычислим *nke* - число равных *ke* элементов в цикле

```
nke = 0
do i = 1, size(a)
  if(a(i) == ke) nke = nke + 1
end do
print *, nke                              !     28
```

Оказывается, что *nke* = 28, а не 21, как это было найдено выше. Дело в том, что BSEARCHQQ в общем случае находит в упорядоченном векторе не первый равный заданному числу элемент, а один из последующих. Чтобы вернуться к первому искомому элементу, в программу после вызова BSEARCHQQ нужно добавить код

```
if(k > 1) then                  ! Если в векторе есть элемент, равный ke,
  it = k                        ! выполним перемещение назад, корректируя
  do k = it, 1, -1              ! величину k - позицию первого равного ke элемента
    if(a(k) /= ke) exit
  end do
  k = k + 1
end if
```

6.10. Справочные функции для любых типов

ALLOCATED(*array*) - возвращает .TRUE., если память выделена под массив *array*, .FALSE. - если не выделена. Параметр *array* должен иметь атрибут ALLOCATABLE. Результат будет неопределенным, если не определен статус массива *array*. Результат имеет стандартный логический тип.

Пример:

```
real, allocatable :: a(:)
...
if(.not. allocated(a)) allocate(a(10))
```

ASSOCIATED(*pointer* [, *target*]). Параметр *pointer* должен быть ссылкой. Состояние привязки *pointer* не должно быть неопределенным. Если параметр *targe*t опущен, то функция ASSOCIATED возвращает .TRUE., если ссылка *pointer* прикреплена к какому-либо адресату. Если параметр *target* задан и имеет атрибут TARGET, то результат равен .TRUE., если ссылка *pointer* прикреплена к *target*. Если *target* задан и имеет атрибут POINTER, то результат равен .TRUE., если как *pointer*, так и *target* прикреплены к одному адресату. При этом состояние привязки ссылки *target* не должно быть неопределенным. Во всех других случаях результат равен .FALSE. Результат имеет стандартный логический тип. В случае массивов .TRUE. возвращается, если совпадают формы аргументов и если элементы ссылки в порядке их следования прикреплены к соответствующим элементам адресата.

Пример:

```
real, pointer :: a(:), b(:), c(:)
real, target :: e(10)
a => e                          ! Назначение ссылки
b => e
c => e(1:10:2)                  ! Ссылка прикрепляется к сечению массива
print *, associated(a, e)       ! T
```

```
print *, associated(a, b)          ! T
print *, associated(c)             ! T
print *, associated(c, e)          ! F
print *, associated(c, e(1:10:2))  ! T
```

PRESENT(*a*) - определяет, задан ли необязательный формальный параметр *a* при вызове процедуры. Функция PRESENT может быть вызвана только в процедуре с необязательными параметрами. Функция возвращает .TRUE., если в вызове процедуры присутствует фактический параметр, соответствующий формальному параметру *a*. В противном случае PRESENT возвращает .FALSE.. Результат имеет стандартный логический тип.

Пример:

```
call who(1, 2)                ! Напечатает:   a present
                              !               b present
call who(1)                   ! Напечатает:   a present
call who(b = 2)               ! Напечатает:   b present
call who( )                   ! Напечатает:   No one
contains
  subroutine who(a, b)
   integer(4), optional :: a, b
   if(present(a)) print *, 'a present'
   if(present(b)) print *, 'b present'
   if(.not. present(a) .and. .not. present(b)) print *, 'No one'
  end subroutine who
end
```

KIND(*x*) - возвращает стандартное целое, равное значению параметра разновидности аргумента *x*.

Пример:

```
real(kind(1e0)), parameter :: one = 1.0     ! Вещественные константа one
real(kind(1e0)) :: err                      ! и переменная err одинарной точности
```

6.11. Числовые справочные и преобразовывающие функции

6.11.1. Модели данных целого и вещественного типа

Каждая разновидность целого и вещественного типа содержит конечное множество чисел. Так, тип INTEGER(2) представляет все целые числа из диапазона от -32,768 до +32,767. Каждое такое множество чисел может быть описано моделью. Данные о параметрах модели заданной разновидности типа и о конкретных характеристиках числа в задающей его модели позволяют получать встроенные числовые справочные и преобразовывающие функции, а также функции IMACH, AMACH и DMACH библиотеки IMSL.

Двоичное представление целого числа *i* задается формулой

$$i = \underbrace{(-1)^s}_{\text{знак}} b_0 b_1 \ldots b_{Q-1},$$

где s - это 0 или 1 (+ или -); b_i - двоичное число (0 или 1); Q - число цифр в целом числе по основанию 2.

Вещественные числа с плавающей точкой представляются в CVF и FPS в близком соответствии со стандартом IEEE для арифметики с плавающей точкой (ANSI/IEEE Std 754-1985, IEEE Standard for Binary Floating-Point Arithmetic, 1985). Фортран поддерживает форматы одинарной точности - тип REAL(4), двойной точности - тип REAL(8) и повышенной точности, используемой для выполнения промежуточных операций. Например, в следующем коде:

```
real(4) :: a, b, c, d, f                ! Задана опция /Op компилятора FPS
b = 0.0; c = 1.0e30; d = 1.0e30; f = 1.0e-30
a = (b + (c * d)) / 2.0 * f
print *, a                              ! 5.000000E+29
```

промежуточные вычисления, если задана опция компилятора /Op, FPS выполнит с повышенной точностью. Если же при компиляции задана опция /Oxp, предусматривающая полную оптимизацию скорости вычислений и проверку ошибок, то на этапе компиляции возникнут предупреждения вида (по причине переполнения в результате умножения c * d = 1.0e30 * 1.0e30)

```
warning F4756: overflow in constant arithmetic
```

а результатом вычислений будет машинная бесконечность - 1#INF.....

Двоичное представление вещественного числа x с плавающей точкой задается формулой

$$x = \underbrace{(-1)^s}_{\text{знак}} \underbrace{b_0.b_1 b_2 \ldots b_{P-1}}_{\text{мантисса}} \times 2^E,$$

где s - это 0 или 1 (+ или -); b_i - двоичное число (0 или 1); P - число цифр в мантиссе нормализованного представления вещественного числа по основанию 2; E - целое число, называемое (*двоичным*) *порядком*, из отрезка $E_{\min} \le E \le E_{\max}$. В табл. 6.1 приводятся значения параметров модели вещественных чисел для одинарной, двойной и повышенной точности.

Таблица 6.1. Параметры модели чисел стандарта IEEE Std 754

Параметр	*Точность*		
	одинарная	*двойная*	*повышенная*
Число бит для знака	1	1	1
P	24	53	64
$E_{\max}$	+128	+1024	+16384

E_{min}	-125	-1021	-16381
Смещение двоичного порядка	+126	+1022	+16382
Число бит для двоичного порядка	8	11	15
Число бит для числа	32	64	80

Стандарт требует, чтобы числа одинарной и двойной точности представлялись в нормализованном виде, поэтому всегда $b_0 = 1$ и, следовательно, для представления мантиссы чисел одинарной точности требуется 23 бита, а двойной - 52. Числа повышенной точности не нормализуются, поэтому для мантиссы требуется 64 бита. При записи порядка к нему с целью повышения скорости вычислений добавляется *смещение*, поэтому порядок всегда представляется в виде положительного числа *e*. Реально, однако, порядок $E = e$ - *смещение*. Структура формата вещественных чисел двойной точности с плавающей точкой приведена на рис. 6.1.

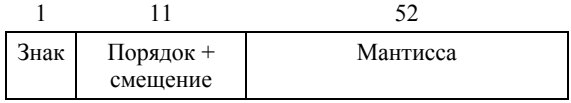


Рис. 6.1. Структура IEEE-формата вещественных чисел двойной точности

Замечания:

1. Множество представимых в компьютере чисел с плавающей точкой конечно. Так, для типа REAL(4) их число примерно равно 2^{31}.

2. FPS и CVF содержат программу BitViewer просмотра двоичного представления вещественных чисел одинарной и двойной точности.

6.11.2. Числовые справочные функции

Эти функции выдают характеристики модели, в которой содержится параметр функции. Параметром функции может быть как скаляр, так и массив. Значение параметра может быть неопределенным.

DIGITS(*x*) - возвращает число двоичных значащих цифр в модели представления *x* (т. е. *Q* или *P*). Параметр *x* может быть целого или вещественного типа. Результат имеет стандартный целый тип.

Тип параметра x	*DIGITS(x)*
INTEGER(1)	7
INTEGER(2)	15
INTEGER(4)	31
REAL(4)	24

REAL(8)	53

EPSILON(x) - возвращает 2^{1-P}:

Тип параметра x	*EPSILON(x)*
REAL(8)	2.22044049250313E-016
REAL(4)	1.192093E-07

Замечание. Возвращаемое функцией и EPSILON число часто называют *машинной точностью* и обозначают $\varepsilon_м$.

HUGE(x) - для целого или вещественного x возвращает наибольшее значение x. Тип и параметр типа результата такие же, как у x. Значение равно $2^Q - 1$ - для целого x и $\left(1-2^{-P}\right)2^{E_{\max}}$ - для вещественного x.

Тип параметра x	*HUGE(x)*
INTEGER(1)	127
INTEGER(2)	32,767
INTEGER(4)	2,147,483,647
REAL(4)	3.402823E+38
REAL(8)	1.797693134862316E+308

MAXEXPONENT(x) - для вещественного x возвращает максимальное значение порядка, т. е. $E_{\max}$. Результат функции имеет стандартный целый тип.

Тип параметра x	*MAXEXPONENT(x)*
REAL(4)	128
REAL(8)	1024

MINEXPONENT(x) - для вещественного x возвращает минимальное значение порядка, т. е. $E_{\min}$. Результат функции имеет стандартный целый тип.

Тип параметра x	*MINEXPONENT(x)*
REAL(4)	-125
REAL(8)	-1021

PRECISION(x) - для вещественного или комплексного x возвращает число значащих цифр, следующих после десятичной точки, используемых для представления чисел с таким же параметром типа, как у x. Результат функции имеет стандартный целый тип.

Тип параметра x	*PRECISION(x)*

REAL(4) или COMPLEX(4)	6
REAL(8) или COMPLEX(8)	15

RADIX(*x*) - для целого или вещественного *x* возвращает стандартное целое, равное основанию системы счисления, используемой для представления чисел. Например:

```
print *, radix(0.0)                ! 2
```

RANGE(*x*) - для целого, вещественного или комплексного *x* возвращает эквивалентный десятичный степенной диапазон значений *x*, т. е.

INT(LOG10(*huge*))

для целого и

INT(MIN(LOG10(*huge*), -LOG10(*tiny*)))

для вещественного или комплексного *x*, где *huge* и *tiny* - наибольшее и наименьшее числа в соответствующих разновидностях типа. Результат функции - стандартного целого типа.

Тип параметра x	*RANGE(x)*
INTEGER(1)	2
INTEGER(2)	4
INTEGER(4)	9
REAL(4) или COMPLEX(4)	37
REAL(8) или COMPLEX(8)	307

TINY(*x*) - для вещественного *x* возвращает наименьшее положительное значение *x*, т. е. $2^{E_{\min}}$. Тип и параметр типа результата такие же, как у *x*.

Тип параметра x	*TINY(x)*
REAL(4)	1.175494E-38
REAL(8)	2.225073858507201E-308

6.12. Элементные функции получения данных о компонентах представления вещественных чисел

Функции такого рода возвращают значение, связанное с компонентами модельного представления фактического значения аргумента.

EXPONENT(*x*) - возвращает степенную часть (т. е. порядок *E*) двоичного представления заданного вещественного *x*. Результат - стандартного целого типа. Результат равен нулю, если *x* = 0. Например:

```
real(4) :: r1 = 1.0, r2 = 123456.7
real(8) :: r3 = 1.0d0, r4 = 123456789123456.7
```

```
write(*,*) exponent(r1)          !        1
write(*,*) exponent(r2)          !       17
write(*,*) exponent(r3)          !        1
write(*,*) exponent(r4)          !       47
```

FRACTION(*x*) - возвращает мантиссу - дробную часть двоичного представления *x*, т. е. $2^{-E}x$. Тип *x* - вещественный. Тип и разновидность типа результата такие же, как у *x*.

Например:

```
print *, fraction(3.0)           !      0.75
print *, fraction(1024.0)        !      0.5
```

NEAREST(*x*, *s*) - возвращает вещественное значение с таким же параметром типа, как у *x*, равное ближайшему к *x* машинному числу, большему *x*, если *s* > 0, и меньшему *x*, если *s* < 0; *s* не может равняться нулю. Например:

```
real(4) :: r1 = 3.0
real(8) :: r2 = 3.0_8
! Используем для вывода шестнадцатеричную систему счисления
write(*, '(1x,z18)') nearest (r1, 2.0)          ! 40400001
write(*, '(1x,z18)') nearest (r1, -2.0)         ! 403FFFFF
write(*, '(1x,z18)') nearest (r2, 2.0_8)        !4008000000000001
write(*, '(1x,z18)') nearest (r2, -2.0_8)       !4007FFFFFFFFFFF
```

Замечание. Числа с плавающей точкой между нулем и HUGE(*x*) распределены неравномерно. В случае REAL(4) между каждыми соседними степенями двойки находится примерно 2^{22} чисел с плавающей точкой. Так, примерно 2^{22} чисел находится между 2^{-125} и 2^{-124} и столько же между 2^{124} и 2^{125}. Простое сопоставление говорит о том, что числа с плавающей точкой гуще расположены вблизи нуля.

RRSPACING(*x*) - возвращает вещественное значение с таким же параметром типа, как у *x*, равное обратной величине относительного расстояния между числами в двоичном представлении *x*, в области, близкой к *x*, т. е. $2^{P}\left|2^{-E}x\right|$.

```
print *, rrspacing( 3.0_4)       !      1.258291e+07
print *, rrspacing(-3.0_4)       !      1.258291e+07
```

SCALE(*x, i*) - возвращает вещественное значение с таким же параметром типа, как у *x*, равное $2^{i}x$, где *i* - целое число.

```
print *, scale(5.2, 2)           !      20.800000
```

SET_EXPONENT(*x, i*) - возвращает вещественное значение с таким же параметром типа, как у *x*, равное $2^{i-E}x$, где *i* - целое число, а *E* = EXPONENT(*x*).

SPACING(*x*) - возвращает вещественное значение с таким же параметром типа, как у *x*, равное абсолютному расстоянию между числами в двоичном представлении, в области, близкой к *x*, т. е. 2^{P-E}.

```
print *, spacing(3.0_4)          !     2.384186e-07
print *, spacing(-3.0_4)         !     2.384186e-07
```

6.13. Преобразования для параметра разновидности

Следующие две функции возвращают минимальное значение параметра разновидности, удовлетворяющее заданным критериям. Аргументы и результаты функций - скаляры. Тип результата - стандартный целый.

SELECT_INT_KIND(*r*) - возвращает значение параметра разновидности целого типа, в котором содержатся все целые числа интервала $-10^r < n < 10^r$. При наличии более одной подходящей разновидности выбирается наименьшее значение параметра разновидности. Результат равен -1, если ни одна из разновидностей не содержит все числа интервала, задаваемого аргументом *r*.

```
print *, selected_int_kind(8)          !     4
print *, selected_int_kind(3)          !     2
print *, selected_int_kind(10)         !     -1 (нет подходящей разновидности)
```

SELECTED_REAL_KIND([*p*] [, *r*]) - возвращает значение параметра разновидности вещественного типа, в котором содержатся все вещественные числа интервала $-10^r < x < 10^r$, десятичная точность которых не хуже *p*. Не допускается одновременное отсутствие двух аргументов. При наличии более одной подходящей разновидности выбирается разновидность с наименьшей десятичной точностью. Функция возвращает -1, если недоступна требуемая точность. Возвращает -2, если недоступен требуемый десятичный степенной диапазон. Возвращает -3, если недоступно и то и другое. Например:

```
kp = 0
do while(selected_real_kind(p = kp) > 0)
 kp = kp + 1
end do
kr = 300
do while(selected_real_kind(r = kr) > 0)
 kr = kr + 1
end do
print *, selected_real_kind(p = kp), kp      !     -1          16
print *, selected_real_kind(r = kr), kr      !     -2         308
print *, selected_real_kind(kp, kr)          !     -3
end
```

6.14. Процедуры для работы с битами

Встроенные процедуры работают с битами, которые содержатся в машинном представлении целых чисел. В основе процедур лежит модель,

согласно которой целое число содержит *s* бит со значениями w_k, $k = 0, 1, ..., s - 1$. Нумерация битов выполняется справа налево: самый правый бит имеет номер 0, а самый левый - *s* - 1. Значение w_k *k*-го бита может равняться либо нулю, либо единице.

6.14.1. Справочная функция BIT_SIZE

BIT_SIZE(*i*) - возвращает число бит, необходимых для представления целых чисел с такой же разновидностью типа, как у аргумента. Результат имеет тот же параметр типа, что и аргумент.

Тип i	*BIT_SIZE(i)*
INTEGER(1)	8
INTEGER(2)	16
INTEGER(4)	32

6.14.2. Элементные функции для работы с битами

BTEST(*i*, *pos*) - возвращает стандартную логическую величину, равную .TRUE., если бит с номером *pos* целого аргумента *i* имеет значение 1, и .FALSE. - в противном случае. Аргумент *pos* должен быть целого типа и иметь значение в интервале $0 \le pos <$ BIT_SIZE(*i*).

Пример:

```
integer(1) :: iarr(2) = (/ 2#10101010, 2#11010101 /)
logical result(2)
result = btest(iarr, (/ 0, 0 /))                    ! F T
write(*, *) result
write(*, *) btest(2#0001110001111000, 2)            ! F
write(*, *) btest(2#0001110001111000, 3)            ! T
```

IAND(*i, j*) - возвращает логическое И между соответствующими битами аргументов *i* и *j*: устанавливает в *k*-й разряд результата 1, если *k*-й разряд первого и второго параметров равен единице. В противном случае в *k*-й разряд результата устанавливается 0. Целочисленные аргументы *i* и *j* должны иметь одинаковые параметры типа. Тот же параметр типа будет иметь и результат.

IBCHNG(*i, pos*) - возвращает целый результат с таким же параметром типа, как у *i*, и значением, совпадающим с *i*, за исключением бита с номером *pos*, значение которого заменяется на противоположное. Аргумент *pos* должен быть целым и иметь значение в интервале $0 \le pos <$ BIT_SIZE(*i*).

IBCLR(*i, pos*) - возвращает целый результат с таким же параметром типа, как у *i*, и значением, совпадающим с *i*, за исключением бита с номером *pos*, который обнуляется. Аргумент *pos* должен быть целым и иметь значение в интервале $0 \le pos <$ BIT_SIZE(*i*).

IBITS(*i, pos, len*) - возвращает целый результат с таким же параметром типа, как у *i*, и значением, равным *len* битам аргумента *i*, начиная с бита с номером *pos*; после смещения этой цепочки из *len* бит вправо и обнуления всех освободившихся битов. Аргументы *pos* и *len* должны быть целыми и иметь неотрицательные значения, такие, что *pos* + *len* ≤ BIT_SIZE(*i*). Например:

```
k = ibits(2#1010, 1, 3)          ! Возвращает 2#101 = 5
print '(b8)', k                  ! 101
```

IBSET(*i, pos*) - возвращает целый результат с таким же параметром типа, как у *i*, и значением, совпадающим с *i*, за исключением бита с номером *pos*, в который устанавливается единица. Аргумент *pos* должен быть целым и иметь значение в интервале 0 ≤ *pos* < BIT_SIZE(*i*).

IEOR(*i, j*) - возвращает логическое исключающее ИЛИ между соответствующими битами аргументов *i* и *j*: устанавливает в *k*-й разряд результата 0, если *k*-й разряд первого и второго параметров равен или единице, или нулю. В противном случае в *k*-й разряд результата устанавливается единица. Целочисленные аргументы *i* и *j* должны иметь одинаковые параметры типа. Тот же параметр типа будет иметь и результат.

IOR(*i, j*) - возвращает логическое ИЛИ между соответствующими битами аргументов *i* и *j*: устанавливает в *k*-й разряд результата единицу, если *k*-й разряд хотя бы одного параметра равен единице. В противном случае в *k*-й разряд результата устанавливается 0. Целочисленные аргументы *i* и *j* должны иметь одинаковые параметры типа. Тот же параметр типа будет иметь и результат.

Пример:

```
integer(2) :: k = 198          !   198     ( = 2#11000110 )
integer(2) :: mask = 129       !   129     ( = 2#10000001 )
write(*, *) iand(k, mask)      !   128     ( = 2#10000000 )
write(*, *) ieor(k, mask)      !    71     ( = 2#01000111 )
write(*, *) ior(k, mask)       !   199     ( = 2#11000111 )
```

ISHA(*i, shift*) - (арифметический сдвиг) возвращает целый результат с таким же параметром типа, как у *i*, и значением, получаемым в результате сдвига битов параметра *i* на *shift* позиций влево (или на -*shift* позиций вправо, если значение *shift* отрицательно). Освобождающиеся при сдвиге влево биты обнуляются, а при сдвиге вправо заполняются значением знакового бита. Аргумент *shift* должен быть целым и удовлетворять неравенству |*shift*| ≤ BIT_SIZE(*i*).

ISHC(*i, shift*) - (циклический сдвиг) возвращает целый результат с таким же параметром типа, как у *i*, и значением, получаемым в результате циклического сдвига всех битов параметра *i* на *shift* позиций влево (или на -*shift* позиций вправо, если значение *shift* отрицательно). Циклический сдвиг выполняется без потерь битов: вытесняемые с одного конца биты

появляются в том же порядке на другом. Аргумент *shift* должен быть целым, причем |*shift*| ≤ BIT_SIZE(*i*).

ISHFT(*i, shift*) - (логический сдвиг) возвращает целый результат с таким же параметром типа, как у *i*, и значением, получаемым в результате сдвига битов параметра *i* на *shift* позиций влево (или на -*shift* позиций вправо, если значение *shift* отрицательно). Освобождающиеся биты как при левом, так и при правом сдвиге обнуляются. В отличие от арифметического сдвига, сдвигается и знаковый разряд. Аргумент *shift* должен быть целым и удовлетворять неравенству |*shift*| ≤ BIT_SIZE(*i*).

ISHTC(*i, shift* [*, size*]) - возвращает целый результат с таким же параметром типа, как у *i*, и значением, получаемым в результате циклического сдвига *size* младших (самых правых) битов параметра *i* (или всех битов, если параметр *size* опущен) на *shift* позиций влево (или на -*shift* позиций вправо, если значение *shift* отрицательно). Аргументы *shift* и *size* должны быть целыми, причем 0 < *size* ≤ BIT_SIZE(*i*); |*shift*| ≤ *size* или |*shift*| ≤ BIT_SIZE(*i*), если опущен параметр *size*.

ISHL(*i, shift*) - выполняет те же действия, что и функция ISHFT.

Пример:

```
integer(1) :: k = -64                !                  -64     ( = 2#11000000 )
integer(1) :: i = 10                 !                   10     ( = 2#00001010 )
integer(2) :: j = 10                 !                   10     ( = 2#0000000000001010 )
! Функция ISHA (правый сдвиг)
print '(1x, b8.8)', isha(k, -3)      !             11111000     ( = -8 )
! Функция ISHL (правый сдвиг)
print '(1x, b8.8)', ishl(k, -3)      !             00011000     ( = 24 )
! Функция ISHC (левый сдвиг)
print '(1x, b8.8)', ishc(i, 5)       !             01000001     ( = 65 )
! Функция ISHFT (тот же результат выдадут функции ISHA и ISHL)
print '(1x, b8.8)', ishft(i, 5)      !             01000000     ( =  64 )
! Функция ISHFTC
print '(1x, b8.8)', ishftc(i, 2, 3)  !             00001001     ( = 9 )
print '(1x, b8.8)', ishftc(i, -2, 3) !             00001100     ( = 12 )
print '(1x, b16.16)', ishftc(j, 2, 3) !  0000000000001001     ( = 9 )
```

NOT(*i*) - (логическое дополнение) возвращает целый результат с таким же параметром типа, как у *i*. Бит результата равен единице, если соответствующий бит параметра *i* равен нулю, и, наоборот, бит результата равен нулю, если соответствующий бит параметра *i* равен единице. Например, NOT(2#1001) возвращает 2#0110.

6.14.3. Элементная подпрограмма MVBITS

CALL MVBITS(*from, frompos, len, to, topos*) - копирует из *from* последовательность из *len* бит, начиная с бита номер *frompos*, в *to*, начиная с бита номер *topos*. Остальные биты в *to* не меняются. Отсчет позиций

выполняется начиная с самого правого бита, номер которого равен нулю. Параметры *from, frompos, len* и *topos* имеют целый тип и вид связи IN: *to* - параметр целого типа с видом связи INOUT. Параметры должны удовлетворять условиям: *len* ≥ 0, *frompos* + *len* ≤ BIT_SIZE(*from*), *frompos* ≥ 0, *topos* ≥ 0, *topos* + *len* ≤ BIT_SIZE(*to*). Параметры *from* и *to* должны иметь одну разновидность типа. В качестве *from* и *to* можно использовать одну и ту же переменную.

Пример:

```
integer(1) :: iso = 13          ! 2#00001101
integer(1) :: tar = 6           ! 2#00000110
call mvbits(iso, 2, 2, tar, 0)  ! Возвращает tar = 00000111
```

6.14.4. Пример использования битовых функций

Достаточно часто поразрядные операции находят применение в программах машинной графики. Рассмотрим в качестве примера применяемый в машинной графике для решения задачи отсечения алгоритм Сазерленда - Кохена. Задача отсечения заключается в удалении элементов изображения, лежащих вне заданной границы (рис. 6.2). Возьмем задачу, в которой границей области является прямоугольник, который далее мы будем называть *окном вывода*. Задача отсечения многократно решается при работе с изображением для большого числа отрезков, поэтому важно, чтобы алгоритм ее решения обладал высоким быстродействием.

В машинной графике экран монитора отображается на растровую плоскость, размеры которой зависят от возможностей видеоадаптера и монитора. Координаты на растровой плоскости целочисленные. Учитывая обеспечиваемое видеоадаптерами разрешение, например 800 * 600 или 1024 * 768 пикселей, для координат используется тип INTEGER(2).

В алгоритме Сазерленда - Кохена взаимное расположение отрезка и окна вывода определяется так. Окно вывода разбивает своими сторонами и их продолжениями растровую плоскость на 9 областей.

Установим для каждой области 4-битовый код (рис. 6.2), в котором:

- единица в бите 0 означает, что точка лежит ниже окна вывода;
- единица в бите 1 означает, что точка лежит правее окна вывода;
- единица в бите 2 означает, что точка лежит выше окна вывода;
- единица в бите 3 означает, что точка лежит левее окна вывода.

Пусть 1 и 2 - номера вершин отрезка; *c*1 и *c*2 - коды областей расположения соответственно первой и второй вершины. XL, XR, YB, YT - координаты границ окна вывода. Очевидно, что

- отрезок расположен внутри окна, если IOR(*c*1, *c*2) возвращает 0;
- отрезок не пересекает окно, если IAND(*c*1, *c*2) > 0;
- отрезок может пересекать окно, если IAND(*c*1, *c*2) возвращает 0 и IOR(*c*1, *c*2) > 0.

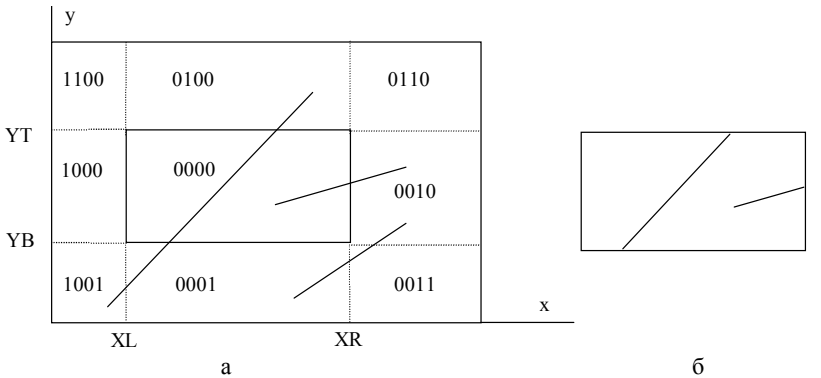


Рис. 6.2. Задача отсечения: а - коды областей; б - решение задачи отсечения

Для сокращения кода программы сначала выполним все отсечения для вершины 1, а затем поменяем вершины 1 и 2 местами (вершине 2 дадим номер 1; вершине 1 - номер 2) и вновь продолжим исследование вершины 1.

Алгоритм:

Интерфейс:

Входные данные:
XL, XR - *x*-координаты правой и левой границ окна вывода;
YB, YT - *y*-координаты нижней и верхней границ окна вывода;
*x*1, *y*1 и *x*2, *y*2 - координаты вершин отрезка.

Выходные данные:
*x*1, *y*1 и *x*2, *y*2 - координаты вершин усеченного отрезка (или исходного, если отрезок целиком принадлежит окну вывода) после решения задачи отсечения. В случае расположения отрезка вне окна вывода выведем сообщение "Отрезок вне окна".

Промежуточные данные:
*c*1, *c*2 - коды областей расположения первой и второй вершин отрезка;
*x*1, *y*1 и *x*2, *y*2 - координаты вершин отрезка на линиях отсечения;
fl - есть истина, если выполнен обмен вершинами 1 и 2.

Алгоритм:

1°. Начало.
2°. Вычислить c1, c2.
3°. Если отрезок может пересекать окно вывода, то перейти к п. 4°, иначе перейти к п. 6° (все отсечения для c1 выполнены).
4°. Если c1 равен нулю, то поменять местами вершины 1 и 2.
5°. Найти линию отсечения, с которой пересекается отрезок 1, и перенести вершину 1 в точку пересечения отрезка и линии отсечения; вычислить c1 и перейти к п. 3°.

6°. Если IOR(*c*1, C2) = 0, то отрезок внутри окна;
вывести *x*1, *y*1, *x*2, *y*2, иначе отрезок вне окна.
7°. Конец.

Программа содержит две процедуры: подпрограмму *swap* - обмена значениями переменных - и функцию *code*, возвращающую код области расположения вершины отрезка.

```
program clip                              ! Текст программы отсечения
integer(2) :: XL = 15, XR = 60, YB = 15, YT = 60
integer(2) :: x1 = 10, y1 = 10, x2 = 65, y2 = 65
integer(2) c1, c2, code
logical :: fl = .false.
c1 = code(x1, y1, XL, XR, YB, YT)
c2 = code(x2, y2, XL, XR, YB, YT)
do while(iand(c1, c2) == 0 .and. ior(c1, c2) > 0)
 if(c1 == 0) then
  fl = .not. fl                           ! Обмен вершин отрезка
  call swap(x1, x2);  call swap(y1, y2);  call swap(c1, c2)
 end if
 if(x1 < XL) then                         ! Отсечение слева
  y1 = y1 + dfloat(y2 - y1) * dfloat(XL - x1) / dfloat(x2 - x1)
  x1 = XL
 else if(y1 < YB) then                    ! Отсечение снизу
  x1 = x1 + dfloat(x2 - x1) * dfloat(YB - y1) / dfloat(y2 - y1)
  y1 = YB
 else if(x1 > XR) then                    ! Отсечение справа
  y1 = y1 + dfloat(y2 - y1) * dfloat(XR - x1) / dfloat(x2 - x1)
  x1 = XR
 else if(y1 > YT) then                    ! Отсечение сверху
  x1 = x1 + dfloat(x2 - x1) * dfloat(YT - y1) / dfloat(y2 - y1)
  y1 = YT
 end if
 c1 = code(x1, y1, XL, XR, YB, YT)        ! Код вершины 2 не изменился
end do
if(fl) then                               ! Обмен вершин отрезка
 call swap(x1, x2)                        ! Восстанавливаем прежний порядок
 call swap(y1, y2)                        ! вершин
 call swap(c1, c2)
end if
if(ior(c1, c2) == 0) then
 write(*, 1) x1, y1, x2, y2               ! x1, y1: 15, 15; x2, y2: 60, 60
else
 write(*, *) 'Отрезок вне окна вывода'
end if
1 format(' x1, y1:', i4, ',', i4, ';  x2, y2:', i4, ',', i4)
end program clip
```

```
subroutine swap(a, b)
 integer(2) a, b, hold
 hold = a;  a = b;  b = hold
end subroutine swap

! Вычисление кода области расположения вершины с координатами x, y
function code(x, y, XL, XR, YB, YT) result (vcode)
 integer(2) x, y, XL, XR, YB, YT, vcode
 vcode = 0
 if(x < XL) vcode = ior(vcode, 2#1000)
 if(y < YB) vcode = ior(vcode, 2#0001)
 if(x > XR) vcode = ior(vcode, 2#0010)
 if(y > YT) vcode = ior(vcode, 2#0100)
end function code
```

6.15. Символьные функции

Встроенные символьные функции ADJUSTL, ADJUSTR, LGE, LGT, LLE, LLT, INDEX, LEN_TRIM, REPEAT, SCAN, TRIM, VERIFY позволяют сравнивать символьные выражения, вычислять их длину, осуществлять поиск в строках других подстрок и выполнять преобразования строк. Функции рассмотрены в разд. 3.8.8.

6.16. Процедуры для работы с памятью

LOC(*gen*) - встроенная функция; возвращает машинный адрес аргумента *gen* или адрес временного результата для аргумента *gen*. Результат имеет тип INTEGER(4).

Если параметр *gen* является выражением, вызовом функции или константой, то создается временная переменная, содержащая результат выражения, вызов функции или константу, и функция LOC возвращает адрес этой временной переменной. Во всех других случаях функция возвращает машинный адрес фактического параметра.

MALLOC(*i*) - встроенная функция; выделяет область памяти размером в *i* байт. Функция возвращает начальный адрес выделенной памяти. Тип аргумента и результата функции - INTEGER(4).

CALL FREE(*i*) - встроенная подпрограмма; освобождает выделенную функцией MALLOC область памяти; *i* - начальный адрес выделенной функцией MALLOC памяти. Тип параметра *i* - INTEGER(4).

Предупреждение. Если освобождаемая память не была ранее выделена функцией MALLOC или память освобождается более одного раза, то результат непредсказуем и выполнение подпрограммы FREE может серьезно повредить занимаемую программой память.

Пример:

```
integer(4) addr, size
size = 1024                    ! Размер в байтах
addr = malloc(size)            ! Выделяем память размером size
...
call free(addr)                ! и освобождаем ее
end
```

6.17. Проверка состояния "конец файла"

Встроенная функция EOF(*устройство*) возвращает .TRUE., если файл позиционирован на запись "конец файла" или вслед за этой записью, и .FALSE. - в противном случае. (Рассмотрена в разд. 11.16.)

6.18. Неэлементные подпрограммы даты и времени

CALL DATE_AND_TIME([*date*] [, *time*] [, *zone*] [, *values*]) - возвращает дату и время, которые показывают встроенные системные часы. Все параметры процедуры имеют вид связи OUT.

date - текстовая скалярная переменная длиной не менее восьми символов, содержащая в восьми первых символах дату в виде CCYYMMDD, где CC соответствует веку, YY - году, MM - месяцу, DD - дню.

time - текстовая скалярная переменная длиной не менее 10 символов, содержащая в 10 первых символах время в виде HHMMSS.SSS, где HH соответствует часу, MM - минутам, SS - секундам, SSS - миллисекундам.

zone - текстовая скалярная переменная длиной не менее пяти символов, содержащая в пяти первых символах разницу между местным временем и универсальным согласованным временем (средним временем по Гринвичу) в виде SHHMM, где S - знак (+ или -), HH - часы, MM - минуты.

values - одномерный стандартный целый массив размером не менее восьми, содержащий последовательность значений: год, месяц, день, разница во времени (в минутах) по отношению ко времени по Гринвичу, час дня, минуты, секунды, миллисекунды. Если какое-либо значение недоступно, то соответствующий элемент массива равен HUGE(0).

```
character(10) dat, tim, zon
call date_and_time(date = dat, time = tim, zone = zon)
print *, dat, tim, ' ', zon
```

CALL SYSTEM_CLOCK([*count*] [, *count_rate*] [, *count_max*]) - возвращает текущее значение системного таймера и его характеристики. Все параметры имеют стандартный целый тип. Вид связи параметров - OUT.

count - текущее значение системного таймера или HUGE(0) при его отсутствии. Значение *count* увеличивается на единицу при каждом отсчете

таймера до тех пор, пока не достигнет значения *count_max*. После чего, начиная с нуля, выполняется новый цикл отсчета времени.

count_rate - число отсчетов таймера в секунду или 0, если таймер отсутствует.

count_max - максимальное значение, которое может принимать *count* или 0, если таймер отсутствует.

```
integer cr, cm
call system_clock(count_rate = cr, count_max = cm)
write(*,*) cr, cm                    !       1       86399
```

Подпрограмму DATE_AND_TIME можно приспособить для измерения времени вычислений, создав функцию *timer*:

```
function timer( )                    ! Возвращает процессорное время
 real(8) :: timer                    ! в миллисекундах
 integer(4) :: ival(8)
 call date_and_time(values = ival)
 timer =  dble(ival(8)) * 0.001_8 +                    &
          dble(ival(7)) + dble(ival(6)) * 60.0_8 +     &
          dble(ival(5)) * 3600.0_8
end function timer
```

Пример. Замеряется время вычислений при правильной и неоптимальной организации вложенных циклов.

```
program toappr
 integer(4), parameter :: n = 1000
 real(4), dimension(n, n) :: array1, array2
 real(8) :: start_time1, finish_time1   ! Время начала и завершения вычислений
 real(8) :: start_time2, finish_time2   ! соответственно при правильной и
                                        ! неоптимальной организации вложенных циклов
 real(8) :: timer                       ! Функция, возвращающая процессорное время
                                        ! в миллисекундах; имеет тип REAL(8)
 array1 = 1.1; array2 = 1.5             ! Инициализация массивов
! Правильная организация вложенного цикла обеспечивает естественный доступ
! к элементам массива по столбцам
 start_time1 = timer( )                 ! Начало вычислений
 do j = 1, n                            ! Правильно: сначала задается столбец,
  do i = 1, n                           ! а затем меняется индекс строки
   array1(i, j) = array1(i, j) + array2(i, j) * 3.3
  end do
 end do
 finish_time1 = timer( )                ! Конец вычислений
 array1 = 1.1; array2 = 1.5             ! Повторная инициализация массивов
! Неоптимальная организация вложенного цикла - неестественный доступ
! к элементам массива по строкам
 start_time2 = timer( )                 ! Начало вычислений
 do i = 1, n                            ! Так программировать не надо. Обеспечьте
  do j = 1, n                           ! доступ к элементам массива по столбцам
```

```
  array1(i, j) = array1(i, j) + array2(i, j) * 3.3
 end do
end do
finish_time2 = timer( )                    ! Конец вычислений
print *, (finish_time1 - start_time1) / (finish_time2 - start_time2)
! Результат:      0.629629629613660
end program toappr
```

6.19. Случайные числа

Генерация случайных чисел выполняется подпрограммой RANDOM_NUMBER от значений, содержащихся в затравочном массиве. Размер и значения этого массива возвращаются подпрограммой RANDOM_SEED. Она же позволяет и изменить затравку.

CALL RANDOM_NUMBER(*harvest*) - возвращает псевдослучайное число *harvest* или массив *harvest* таких чисел из равномерно распределенного интервала: $0 \le x < 1$. Тип *harvest* - стандартный вещественный. Вид связи параметра *harvest* - OUT.

Стартовая (затравочная) точка для генератора случайных чисел устанавливается и может быть запрошена RANDOM_SEED. Если RANDOM_SEED не использована, то значение затравки зависит от процессора.

```
real x, hav(3)
call random_seed( )
call random_number(x)
call random_number(hav)
print *, hav                          !  5.252978E-01   6.502543E-01   4.247389E-01
```

CALL RANDOM_SEED([*size*] [, *put*] [, *get*]) - изменяет стартовую точку (затравку) генератора псевдослучайных чисел, используемого подпрограммой RANDOM_NUMBER.

size - стандартное целое число, имеющее вид связи OUT, равное размеру *n* создаваемого процессором затравочного массива.

put - стандартный целый массив с видом связи IN, используемый процессором для изменения затравки.

get - стандартный целый массив с видом связи OUT, в который заносятся текущие значения затравки.

При вызове RANDOM_SEED должно быть задано не более одного параметра. Размеры *put* и *get* должны быть больше, чем размер массива, который используется процессором для хранения затравочных чисел. Этот размер можно определить, вызвав RANDOM_SEED с параметром *size*. В настоящей реализации *size* = 1.

Если при вызове RANDOM_SEED не задано ни одного параметра, то процессор устанавливает стартовую точку генератора случайных чисел в зависимости от системного времени.

Пример:

```
integer sv, k
integer, allocatable :: ap(:), ag(:)
call random_seed(size = sv)       ! Читаем размер затравочного массива
allocate(ap(sv), ag(sv))          ! Размещаем массивы
call random_seed(get = ag)        ! Читаем в массив ag значение затравки
print *, sv                       !       1
print *, (ag(k), k = 1, sv)       !       1
ap = (/ (100*k, k = 1, sv) /)
call random_seed(put = ap)        ! Переопределяем значение затравки
```

Помимо встроенной подпрограммы RANDOM_NUMBER в CVF и FPS есть дополнительные подпрограммы получения случайных чисел RANDOM и RAN, а также функции библиотеки PortLib: DRAND, DRANDM, IRAND, IRANDM, RAN, RAND и RANDOM. Все они взаимозаменяемы, поскольку используют один и тот же алгоритм генерации псевдослучайных чисел.

6.20. Встроенная подпрограмма CPU_TIME

Подпрограмма CPU_TIME(*time*) возвращает процессорное время *time*, тип которого - REAL(4). Единицы измерения времени - секунды; после десятичной точки *time* содержит две значащих цифры. Может быть использована для оценки продолжительности вычислений, например:

```
real(4) :: start_time, finish_time
call cpu_time(start_time)
<Вычисления>
call cpu_time(finish_time)
print *, 'Время вычислений = ', finish_time - start_time
```

7. Управляющие операторы и конструкции

Последовательность выполнения программы определяется операторами и конструкциями, обеспечивающими ветвления и циклы. Совместно с ними могут быть использованы операторы перехода и прерывания цикла. Такие операторы и конструкции относятся к управляющим. Их рассмотрение мы начали во 2-й главе, ограничившись, правда, наиболее часто употребляемыми. Теперь же мы дадим полное описание всех управляющих конструкций (кроме приведенных в разд. 4.7 конструкций WHERE и FORALL). При этом часть управляющих операторов и конструкций будет выделена в разряд нерекомендуемых или устаревших, сохраненных в Фортране с целью преемственности с более ранними версиями.

Как и ранее, необязательные элементы операторов заключаются в квадратные скобки. Используемые в главе сокращения введены в разд. 2.2.

7.1. Оператор GOTO безусловного перехода

Используется для передачи управления по метке и имеет вид:

GOTO *метка*

В современном Фортране GOTO, так же как альтернативный выход из подпрограммы и дополнительный вход (ENTRY) в процедуру, следует полностью заменить другими управляющими конструкциями.

Замечания:

1. Запрещается переход внутрь конструкций DO, IF, SELECT CASE и WHERE.

2. В Фортране можно также организовать переход по вычислению (вычисляемый GOTO) и предписанию (назначаемый GOTO). Описание этих операторов приведено в прил. 2.

Пример. GOTO используется для организации повторных действий, т. е. цикла.

```
integer(1) in
10 continue
print *, 'Введите число от 1 до 10: '
read  *, in
if(in >= 1 .and. in <= 10) then
 print *, 'Ваш ввод: ', in
else
 print *, 'Ошибка. Повторите ввод'
 goto 10
```

```
end if
...
```

Выполняющий те же действия фрагмент без GOTO выглядит так:

```
integer(1) in
do
 print *, 'Введите число от 1 до 10: '
 read *, in
 if(in >= 1 .and. in <= 10) then
  print *, 'Ваш ввод: ', in
  exit
 end if
 print *, 'Ошибка. Повторите ввод'
end do
...
```

7.2. Оператор и конструкции IF

Дополнительно по сравнению с более ранними версиями Фортрана в управляющие конструкции введено необязательное имя конструкции. Применение именованных конструкций позволяет создавать хорошо читаемые программы даже при большой глубине вложенности управляющих конструкций.

В приводимых в настоящем разделе операторе и конструкциях IF ЛВ должно быть скаляром, т. е. операндами ЛВ не должны быть массивы или их сечения.

7.2.1. Условный логический оператор IF

IF(ЛВ) оператор

Если истинно ЛВ, то выполняется *оператор*, в противном случае управление передается на последующий оператор программы.

В Фортране существует еще один условный оператор - арифметический IF. Этот оператор относится к устаревшим свойствам Фортрана. Его описание дано в прил. 2.

7.2.2. Конструкция IF THEN END IF

[*имя*:] IF(ЛВ) THEN
БОК
END IF [*имя*]

БОК выполняется, если истинно ЛВ. Если присутствует *имя* конструкции, то оно должно быть и в первом и в последнем операторе конструкции, например:

```
swap: if(x < y) then
  hold = x;  x = y;  y = hold
end if swap
```

Замечание. Если БОК содержит один оператор, то лучше использовать оператор

IF(ЛВ) *оператор*

7.2.3. Конструкция IF THEN ELSE END IF

```
[имя:] IF(ЛВ) THEN
         БОК1
       ELSE [имя]
         БОК2
       END IF [имя]
```

В случае истинности ЛВ выполняется БОК1, и выполняется БОК2, если ЛВ ложно. *Имя* конструкции, если оно задано, должно обязательно присутствовать и перед IF и после END IF.

Пример:

```
if(x**2 + y**2  < r**2) then
 print *, 'Точка внутри круга'
else
 print *, 'Точка вне круга'
end if
```

7.2.4. Конструкция IF THEN ELSE IF

```
[имя:] IF(ЛВ1) THEN
         БОК1
       ELSE IF(ЛВ2) THEN [имя]
         БОК2
       ...
       [ELSE [имя]
         БОКn]
       END IF [имя]
```

В случае истинности ЛВ1 выполняется БОК1 и управление передается на следующий за END IF оператор. Если ЛВ1 ложно, то управление передается на следующий ELSE IF, т. е. вычисляется значение ЛВ2 и, если оно истинно, выполняется БОК2. Если оно ложно, то управление передается на следующий ELSE IF, и т. д. Если ложны все ЛВ, то выполняется следующий за завершающим ELSE БОК*n*. Если завершающий ELSE отсутствует, то управление передается на расположенный за END IF оператор. Число операторов ELSE IF в конструкции может быть произвольным. *Имя* в ELSE и в ELSE IF можно задавать, если его имеют операторы IF и END IF. *Имя*, если оно задано, во всех частях конструкции должно быть одинаковым.

Следует обратить внимание на то, что вся конструкция завершается одним END IF. Ясно, что такая запись более экономна, чем запись, использующая отдельные конструкции IF THEN ELSE END IF, например:

```
IF(ЛВ1) THEN                IF(ЛВ1) THEN
  БО1                         БО1
ELSE                        ELSE IF(ЛВ2) THEN
  IF(ЛВ2) THEN                БО2
    БО2                     ELSE
  ELSE                        БО3
    БО3                     END IF
  END IF
END IF
```

Пример. Найти число положительных, отрицательных и нулевых элементов массива.

```
integer :: a(100), np = 0, ne = 0, nz = 0, ca, i
<Ввод массива a>
do i = 1, 100
 ca = a(i)
 val_3: if(ca > 0) then              ! val_3 - имя конструкции
   np = np + 1
 else if(ca < 0) then
  ne = ne + 1
 else  val_3
  nz = nz + 1
 end if val_3
end do
...
```

Замечание. Подсчет искомых значений можно выполнить, применив встроенную функцию COUNT. Правда, вызвать ее придется два раза:

```
np = count(a > 0)
ne = count(a < 0)
nz = 100 - np - ne
```

7.3. Конструкция SELECT CASE

```
[имя:] SELECT CASE(тест-выражение)
       CASE(СП1) [имя]
         [БОК1]
       [CASE(СП2) [имя]
         [БОК2]]
       ...
       [CASE DEFAULT [имя]
         [БОКn]]
       END SELECT [имя]
```

Тест-выражение - целочисленное, символьное типа CHARACTER(1) или логическое скалярное выражение.

СП - список констант, тип которых должен соответствовать типу *тест-выражения*.

Конструкция SELECT CASE работает так: вычисляется значение *тест-выражения*. Если полученное значение находится в списке СП1, то выполняется БОК1; далее управление передается на следующий за END SELECT оператор. Если значение в СП1 не находится, то проверяется, есть ли оно в СП2, и т. д. Если значение *тест-выражения* не найдено ни в одном списке и присутствует оператор CASE DEFAULT, то выполняется БОК*n*, а далее выполняется расположенный за END SELECT оператор. Если же значение *тест-выражения* не найдено ни в одном списке и CASE DEFAULT отсутствует, то ни один из БОК*i* не выполняется и управление передается на следующий за END SELECT оператор.

Список констант СП может содержать одно значение, или состоять из разделенных запятыми констант, или быть задан как диапазон разделенных двоеточием значений, например 5:10 или 'I':'N'. Левая граница должна быть меньше правой. Если задается диапазон символов, то код первого символа должен быть меньше кода второго. Если опущена левая граница, например :10, то в СП содержатся все значения, меньшие или равные правой границе. И наоборот, если опущена верхняя граница, например 5:, то в СП попадают все значения, большие или равные нижней границе. СП может включать также и смесь отдельных значений и диапазонов. Разделителями между отдельными элементами СП являются запятые, например:

```
case(1, 5, 10:15, 33)
```

Нельзя задать в СП диапазон значений, когда *тест-выражение* имеет логический тип. Каждое значение, даже если оно задано в диапазоне значений, может появляться только в одном СП.

SELECT CASE-конструкции могут быть вложенными. При этом каждая конструкция должна завершаться собственным END SELECT.

Нельзя переходить посредством оператора GOTO или в результате альтернативного возврата из подпрограммы внутрь конструкции SELECT CASE или переходить из одной CASE секции в другую. Попытка такого перехода приведет к ошибке компиляции.

Имя конструкции, если оно задано, обязательно должны иметь операторы SELECT CASE и END SELECT.

Пример. Найти число положительных, отрицательных и нулевых элементов целочисленного массива.

```
integer :: a(100), np = 0, ne = 0, nz = 0, i
<Ввод массива a>
do i = 1, 100
```

```
 select case(a(i))
 case(1:)
  np = np + 1
 case(:-1)
  ne = ne + 1
 case(0)                          ! или: CASE DEFAULT
  nz = nz + 1
 endselect
end do
...
```

7.4. DO-циклы. Операторы EXIT и CYCLE

Простейшая конструкция DO

[*имя*:] DO
БОК
END DO [*имя*]

задает бесконечный цикл. Поэтому БОК должен содержать по крайней мере один оператор, например EXIT [*имя*], обеспечивающий выход из этого цикла. *Имя* конструкции, если оно присутствует, должно появляться в операторах DO и END DO.

Пример. Найти первый отрицательный элемент массива *a*(1:100).

```
i = 1
first_n: do                       ! first_n - имя конструкции DO
 if(a(i) < 0 .or. i == 100) exit first_n
 i = i + 1
end do first_n
if(a(i) >= 0) stop 'В массиве нет отрицательных элементов'
print *, a(i)                     ! Первый отрицательный элемент массива
```

Рекомендуемая форма DO-цикла с параметром:

[*имя*:] DO *dovar* = *start*, *stop* [, *inc*]
БОК
END DO [*имя*]

dovar - целая, вещественная одинарной или двойной точности переменная, называемая *переменной цикла* или *параметром цикла*;

start, stop - целые, вещественные одинарной или двойной точности скалярные выражения, задающие диапазон изменения *dovar* ;

inc - целое, вещественное одинарной или двойной точности скалярное выражение. Значение *inc* не может быть равным нулю. Если параметр *inc* отсутствует, то он принимается равным единице.

Число итераций цикла определяется по формуле

ni = MAX(INT((*stop* - *start* + *inc*) / *inc*), 0),

где MAX - функция выбора наибольшего значения, а функция INT возвращает значение, равное целой части числа.

Если DO-цикл с параметром не содержит операторов выхода из цикла, например EXIT, то БОК выполняется *ni* раз.

После завершения цикла значение переменной цикла *dovar* равно (случай *inc* > 0):

- *dovar_ni* + *inc*, если *stop* ≥ *start* и цикл не содержит операторов выхода из цикла, где *dovar_ni* - значение переменной цикла на последней итерации;
- *dovar_ni*, если *stop* ≥ *start* и цикл досрочно прерван, например оператором EXIT или GOTO, где *dovar_ni* - значение переменной цикла *dovar* в момент прерывания цикла;
- *start*, если *stop* < *start*.

Аналогично определяется значение *dovar* и для случая *inc* < 0.

Порядок выполнения DO-цикла с параметром изложен в разд. 2.2.3.1.

Нельзя изменять значение переменной цикла в теле цикла:

```
do k = 1, 10
  k = k + 2                    ! Ошибка. Попытка изменить значение переменной цикла
end do
```

При первом выполнении оператора DO *dovar* = *start*, *stop*, *inc* вычисляются и запоминаются значения выражений *start*, *stop* и *inc*. Все дальнейшие итерации выполняются с этими значениями. Поэтому если *stop*, *start* или *inc* являются переменными и их значения изменяются в теле цикла, то на работе цикла это не отразится.

Замечание. В DO-цикле с вещественным одинарной или двойной точности параметром из-за ошибок округления может быть неправильно подсчитано число итераций, на что обращается внимание в прил. 2.

Рекомендуемая форма DO WHILE-цикла:

[*имя*:] DO WHILE(ЛВ)
БОК
END DO [*имя*]

Если DO WHILE-цикл не содержит операторов прерывания цикла, то БОК выполняется до тех пор, пока истинно скалярное ЛВ.

DO-цикл, DO-цикл с параметром и DO WHILE-цикл могут быть прерваны операторами GOTO, EXIT и CYCLE, а также в результате выполнения оператора RETURN, обеспечивающего возврат из процедуры.

Оператор

EXIT [*имя*]

передает управление из DO-конструкции на первый следующий за конструкцией выполняемый оператор. Если *имя* опущено, то EXIT

обеспечивает выход из текущего цикла, в противном случае EXIT обеспечивает выход из цикла, *имя* которого присутствует в операторе EXIT.

Оператор

CYCLE [*имя*]

передает управление на начало DO-конструкции. При этом операторы, расположенные между CYCLE и оператором END DO конца цикла, не выполняются. Если *имя* опущено, то CYCLE обеспечивает переход на начало текущего цикла, в противном случае CYCLE обеспечивает переход на начало цикла, *имя* которого присутствует в операторе CYCLE.

Пример. Вычислить сумму элементов массива, значения которых больше пяти, завершая вычисления при обнаружении нулевого элемента.

```
integer a(100), sa, c, i
<Ввод массива a>
sa = 0
do i = 1, 100
  c = a(i)
  if(c == 0) exit                ! Досрочный выход из цикла
  if(c <= 5) cycle               ! Суммирование не выполняется
  sa = sa + c
end do
print *, sa
```

Замечание. С позиций структурного программирования приведенные вычисления лучше выполнить, применив объединение условий и отказавшись от операторов EXIT и CYCLE:

```
sa = 0
i = 1
do while(a(i) /= 0 .and. i <= 100)
  if(a(i) > 5) sa = sa + a(i)
  i = i + 1
end do
```

DO-конструкции могут быть вложенными. Степень вложения неограниченна. Вложенным DO-конструкциям следует давать имена, что повысит их наглядность и позволит в ряде случаев сократить код.

Пример. В трехмерном массиве найти первый отрицательный элемент.

```
integer, parameter :: L = 20, m = 10, n = 5
real, dimension(l, m, n) :: a
< ввод массива a >
loop1: do i = 1, L
 loop2: do j = 1, m
  loop3: do k = 1, n
   if(a(i, j, k) < 0.0) exit loop1          ! Выход из цикла loop1
  end do loop3
```

```
 end do loop2
end do loop1
if(i > L) stop 'В массиве нет отрицательных элементов'
print *, a(i, j, k)
end
```

При работе с DO- и DO WHILE-циклами необходимо помнить:

- переменная DO-цикла с параметром *dovar* не может быть изменена операторами этого цикла;
- не допускается переход внутрь цикла посредством выполнения оператора GOTO или альтернативного возврата из подпрограммы;
- не допускается переход на начало DO-конструкции посредством оператора CYCLE, расположенного за пределами этой конструкции (попытка такого перехода может быть предпринята при работе с именованными вложенными DO-конструкциями);
- если оператор IF, SELECT CASE, WHERE или FORALL появляется внутри цикла, то соответствующий ему оператор END IF, END SELECT, END WHERE или END FORALL должен быть внутри того же цикла.

Замечание. Последние три запрета легче контролировать, если соблюдать при записи программы правило рельефа (разд. 2.5).

При записи DO- и DO WHILE-циклов могут быть использованы метки. Проиллюстрируем эти формы записи на примере вычисления суммы отрицательных элементов массива *a*(1:100).

```
! Вариант 1; цикл завершается пустым оператором CONTINUE
sa = 0
do 21, k = 1, 100                    ! После метки можно поставить запятую
 if(a(k) .lt. 0) sa = sa + a(k)
21 continue

! Вариант 2; вместо CONTINUE используется END DO
sa = 0
do 22 k = 1, 100
 if(a(k) .lt. 0) sa = sa + a(k)
22 end do

! Вариант 3; цикл завершается исполняемым оператором
sa = 0
do 23 k = 1, 100
23 if(a(k) .lt. 0) sa = sa + a(k)

! Использующие цикл DO метка [,] WHILE варианты 4, 5 и 6 имеют те же
! различия, что и варианты 1, 2 и 3

! Вариант 4
k = 1
sa = 0
```

```
do 24, while(k .le. 100)          ! После метки можно поставить запятую
 if(a(k) .lt. 0) sa = sa + a(k)
 k = k + 1
24 end do                         ! Конструкцию завершает метка END DO
! Вариант 5
k = 1
sa = 0
do 25 while(k .le. 100)
 if(a(k) .lt. 0) sa = sa + a(k)
 k = k + 1
25 continue                       ! Конструкцию завершает метка CONTINUE
! Вариант 6
k = 1
sa = 0
do 26 while(k .le. 100)
 if(a(k) .lt. 0) sa = sa + a(k)
! Конструкцию завершает метка исполняемый оператор
26 k = k + 1
```

Замечания:

1. После метки в предложении DO и DO WHILE может стоять запятая, что проиллюстрировано первым и четвертым циклами.

2. Оператор CONTINUE является пустым, ничего не выполняющим оператором и может появляться где угодно среди исполняемых операторов.

3. В случае вложенных DO- или DO WHILE-циклов два или более DO- или DO WHILE-цикла могут иметь общую завершающую метку. Правда, END DO может завершать только один DO- или DO WHILE-цикл.

7.5. Возможные замены циклов

Фортран 90 позволяет в задачах с массивами, где ранее использовались циклы, применять более лаконичные и эффективные средства. К ним относятся:

- встроенные функции для работы с массивами;
- операторы и конструкции WHERE и FORALL (разд. 4.7);
- сечения массивов.

Проиллюстрируем сказанное примерами.

Пример 1. Найти строку матрицы с максимальной суммой элементов.

```
integer, parameter :: m = 10, n = 20
integer kmax(1)                   ! Номер искомой строки матрицы
real a(m, n)
<Ввод матрицы a>
kmax = maxloc(sum(a, dim = 2))    ! Функция возвращает массив
print *, kmax
```

Пример 2. Заменить в векторе *b* все отрицательные элементы нулями.

```
integer, parameter :: n = 100
real b(n)
<Ввод вектора b>
where(b < 0) b = 0                    ! Замена отрицательных элементов нулями
```

Пример 3. Поменять местами в матрице *a* первую и последнюю строки.

```
integer, parameter :: m = 10, n = 20
real, allocatable :: temp(:)
real a(m, n)
<Ввод матрицы a>
allocate(temp(1:n))                   ! Выделяем память под temp
temp = a(1, 1:n)                      ! Запомним строку 1 в массиве temp
a(1, 1:n) = a(m, 1:n)                 ! Пишем в строку 1 строку m
a(m, 1:n) = temp                      ! Пишем в строку m из temp
deallocate(temp)                      ! Освобождаем память
```

Пример 4. Сформировать вектор *c* из отрицательных элементов матрицы *a*.

```
integer, parameter :: m = 5, n = 4
integer k
real, allocatable :: c(:)
real a(m, n)
<Ввод матрицы a>
k = count(a < 0)                      ! Число отрицательных элементов в a
if( k == 0 ) stop 'В матрице a нет отрицательных элементов'
allocate( c(k) )
c = pack(a, a < 0)                    ! Переносим в c отрицательные элементы a
```

Пример 5. Вывести построчно двумерный массив.

```
integer, parameter :: m = 5, n = 3
integer i                             ! Элементы матрицы размещены в памяти по столбцам
integer a(m, n) /  1, 2, 3, 4, 5,                &
                   1, 2, 3, 4, 5,                &
                   1, 2, 3, 4, 5 /
do i = 1, m                           ! Помещаем в список вывода сечение массива
  print '(100i3)', a(i, 1:n)
end do
```

В последнем примере вместо неявного цикла ($a(i, j)$, $j = 1, n$) в списке вывода присутствует сечение массива, которое так же, как и неявный цикл, задает отдельную строку массива. Аналогичным образом сечение массива можно использовать и при вводе.

Однако замены циклов сечениями не всегда возможны. Это прежде всего касается циклов, в которых результат итерации зависит от результата предыдущей итерации. Так, цикл

```
do i = 2, n
 a(i) = a(i - 1) + 5.0
end do
```

нельзя заменить присваиванием

```
a(2:n) = a(1:n - 1) + 5.0
```

или конструкцией

```
forall(i = 2:n) a(i) = a(i - 1) + 5.0
```

7.6. Оператор STOP

Оператор прекращает выполнение программы. Имеет синтаксис:

STOP [*сообщение*]

Сообщение - символьная или целочисленная константа в диапазоне от 0 до 99999. Если *сообщение* отсутствует, то после выполнения оператора выводится строка

STOP - Program terminated.

Если *сообщение* является символьной константой, то:

- выводится *сообщение*;
- программа возвращает 0 в операционную систему.

Если *сообщение* является числом, то:

- выводится предложение Return code *число*. Например, если применен оператор STOP 0400, после завершения программы будет выведено предложение Return code 0400;
- программа возвращает в операционную систему последний значащий байт целого числа (значения от 0 до 255) для использования программой, проверяющей значения статусов выполняемых процессов.

Пример:

```
open(2, file = 'b.txt', status = 'old', iostat = icheck)
if(icheck .ne. 0) then
  stop 'File access denied.'
end if
```

7.7. Оператор PAUSE

Оператор временно приостанавливает выполнение программы и позволяет пользователю выполнить команды операционной системы. Имеет синтаксис:

PAUSE [*сообщение*]

Сообщение - символьная или целочисленная константа в диапазоне от 0 до 99999. Если *сообщение* отсутствует, то после выполнения оператора выводится строка

Please enter a blank line (to continue) or a system command.

После выполнения оператора PAUSE возможны следующие действия:

- если пользователь вводит пробел, то управление возвращается программе;
- если пользователь вводит команду, то выполняется команда и управление возвращается программе. Максимальный размер задаваемой на одной строке команды составляет 128 байт;
- если введено слово COMMAND или command, то пользователь может выполнить последовательность команд операционной системы. Для возврата в программу потребуется ввести EXIT (прописными или строчными буквами).

Пример:

```
character(30) filename
pause 'Enter DIR or press Enter to return.'
read(*, '(a)') filename
open(1, file = filename)
```

Замечание. Стандартом 1995 г. оператор PAUSE исключен из Фортрана. Однако он по-прежнему поддерживается и CVF и FPS.

8. Программные единицы

8.1. Общие понятия

При разработке алгоритма исходная задача, как правило, разбивается на отдельные подзадачи. Процесс выделения подзадач крайне важен. Можно без преувеличения сказать, что квалификация программиста во многом определяется его способностью рационально разбить исходную задачу на фрагменты, которые впоследствии реализуются в виде отдельных программных единиц. В Фортране 77 выделенные фрагменты оформлялись, как правило, в виде главной программы и внешних процедур.

Выделенные фрагменты должны обмениваться данными. В Фортране 77 такой обмен выполнялся через параметры процедур, *common*-блоки и, в случае процедуры-функции, через возвращаемое ею значение. В программной единице BLOCK DATA переменным именованного *common*-блока можно было задать начальные значения.

Фортран 90 и 95, сохраняя все возможности Фортрана 77, дополнительно позволяют:

- создавать модули, содержащие глобальные данные и модульные процедуры;
- создавать внутренние процедуры, расположенные внутри главной программы, внешней или модульной процедуры.

Эти нововведения дополнительно позволяют:

- использовать различным программным единицам объявленные в модуле глобальные данные и имеющиеся в нем процедуры;
- использовать объявленные в модуле данные во всех процедурах этого модуля;
- использовать один и тот же объект данных во внутренней процедуре и в ее носителе.

Таким образом, в Фортране можно создать такие программные единицы, как:

- главная программа;
- модули;
- внешние процедуры;
- внутренние процедуры;
- BLOCK DATA.

Дополнительно к названным в программной единице (кроме BLOCK DATA) можно определить и операторные функции (разд. 8.24).

Модуль - это самостоятельная программная единица, которая может в общем случае содержать объявления данных, *common*-блоков, интерфейсы процедур, *namelist*-группы и модульные процедуры. Все не

объявленные PRIVATE компоненты модуля доступны в других (кроме BLOCK DATA) программных единицах.

В Фортране существует два вида процедур: *подпрограммы* и *функции*.

Подпрограмма - это именованная программная единица, в заголовке которой присутствует оператор SUBROUTINE. Вызов подпрограммы осуществляется по ее имени в операторе CALL или при выполнении задаваемого присваивания.

Функция - это именованная программная единица, вызов которой выполняется по ее имени из выражения. В ее заголовке присутствует оператор FUNCTION. Функция содержит результирующую переменную, получающую вследствие выполнения функции значение, которое затем используется в выражении, содержащем вызов функции. Также функция вызывается и при выполнении задаваемой операции.

В Фортране можно задать *внешние*, *внутренние* и *модульные процедуры*.

Внешняя процедура является самостоятельной программной единицей и существует независимо от использующих ее программных единиц. К любой внешней процедуре можно обратиться из главной программы и любой другой процедуры.

Модули и внешние процедуры могут компилироваться отдельно от использующих их программных единиц.

Внутренняя процедура задается в главной программе, внешней или модульной процедуре. Главная программа или процедура называются *носителями* содержащихся в них внутренних процедур. Обратиться к внутренней процедуре можно только внутри ее носителя. Сами же внутренние процедуры уже не могут содержать в себе других внутренних процедур.

Модульная процедура задается в модуле и доступна, если она не объявлена PRIVATE, в любой использующей модуль программной единице. Модуль также является *носителем* по отношению к заданной в нем модульной процедуре, которая, в свою очередь, может быть носителем определенной в ней внутренней процедуры.

Фортран 90 и 95 в отличие от Фортрана 77 поддерживают рекурсивные вызовы процедур, т. е. такие вызовы, в которых процедура прямо или косвенно обращается сама к себе. Оператор заголовка рекурсивной процедуры содержит префикс RECURSIVE.

8.2. Использование программных единиц в проекте

В создаваемом пользователем проекте могут использоваться:

- встроенные процедуры;
- подключаемые процедуры и модули;
- создаваемые при разработке проекта процедуры и модули.

Встроенные процедуры входят в состав Фортрана и автоматически включаются в исполняемый код при обращении к ним в тексте программы. Примеры встроенных процедур: SIN, ALOG, TRIM.

Подключаемые процедуры и *модули* находятся в ранее созданных и поставляемых с Фортраном библиотеках. Перечень и описание таких модулей и процедур дан в прил. 3.

Также с CVF и FPS поставляются математические библиотеки и библиотеки математической статистики.

Все поставляемые с CVF и FPS процедуры при правильной установке программ по умолчанию доступны для компилятора и компоновщика. Для их использования в программной единице следует выполнить ссылку на модуль, содержащий глобальные данные и интерфейсы процедур. Ссылка осуществляется оператором USE.

Пользователь может создать собственные прикладные библиотеки и использовать хранимые в них процедуры и модули в любом из своих проектов. Для доступа к содержащим объектный код библиотекам пользователю следует указать построителю ее полное имя.

Процедуры и модули проекта содержат свежие решения и на начальных стадиях разработки хранятся в *исходном коде*. Размещаются новые программные единицы, как правило, в разных файлах. Если в файле находится несколько процедур или модулей, то порядок их размещения произвольный. Однако текст модуля должен быть размещен до первой имеющейся на него ссылки. В общем случае в одном файле могут существовать внешние процедуры, модули и главная программа. В то же время осмысленное разбиение программных единиц по файлам, порядок их размещения в каждом из файлов, правильное разделение процедур на внешние, внутренние и модульные существенно облегчает разработку программы и ее последующее сопровождение в процессе эксплуатации.

На последующих стадиях работы над программой часть отлаженных процедур и модулей может храниться в откомпилированном виде (объектном коде), часть - включена в библиотеки, содержащие объектный код программных единиц. Недоработанные программные единицы по-прежнему хранятся в исходном коде.

Создаваемые процедуры и модули реализуют выделенные при разработке алгоритма фрагменты и решают проблемы обмена данными между ними. Фрагмент реализуется в виде процедуры, если он:

- представляет типовую задачу, например поиск экстремума функции;
- представляет обособленную задачу, например В/В данных и контроль ошибок В/В.

Также в виде процедур оформляется повторяющийся в программе более одного раза код. Оформление фрагментов в виде процедур улучшает качество программы, сокращает время ее разработки, отладки и

тестирования. Вопрос выбора типа процедуры (внешней, модульной или внутренней) неразрывно связан с проблемой организации данных.

Процедуры и модули полезны и по другой причине. Часто к работе над большим проектом необходимо привлечь бригаду программистов. Организовать работу бригады можно, лишь поручив каждому участнику реализацию той или иной группы процедур и модулей. Понятно, что предварительно должна быть определена общая структура программы, выделены фрагменты, определены данные, которыми фрагменты обмениваются, и способы обмена данными (ассоциирование параметров, ассоциирование через носитель, *use*-ассоциирование, ассоциирование памяти). Такая предварительная и чрезвычайно важная работа называется *проектированием программы*.

Итак, процедуры и модули:

- позволяют сократить расходы на создание программы;
- улучшают читаемость программы и, следовательно, облегчают ее последующую модификацию;
- приводят к сокращению исходного кода;
- могут быть включены в библиотеки и вызваны из любой программы;
- позволяют разделить работу над программой между разными программистами.

8.3. Работа с проектом в среде DS

В CVF и FPS программа рассматривается как проект. Типы возможных проектов приведены в табл. 8.1.

Таблица 8.1. Типы проектов

Тип проекта	*Особенности*
Консоль (EXE)	Однооконный основной проект без графики
Стандартная графика (EXE)	Однооконный основной проект с графикой
QuickWin-графика (EXE)	Многооконный основной проект с графикой
Windows-приложение (EXE)	Многооконный основной проект с полным графическим интерфейсом и Win32 API-функциями
Статическая библиотека (LIB)	Библиотечные процедуры, подключаемые в EXE-файлы
Динамическая библиотека (DLL)	Библиотечные процедуры, подключаемые в процессе выполнения

Первые 4 типа требуют наличия главной программы. Два последних - библиотечные проекты без главной программы.

Тип проекта задает вид генерируемого кода и некоторые параметры проекта. Например, он определяет опции, которые использует компилятор,

библиотеки, применяемые компоновщиком, задает по умолчанию размещение выходных файлов, константы проекта и т. д. Задание типа проекта выполняется при его создании. Порядок создания проекта в CVF и FPS и некоторые операции с проектами рассмотрены в разд. 1.2.

После создания нового проекта можно сгенерировать две его модификации: Debug и Release. Первая - содержит применяемые в режиме отладки настройки компилятора и компоновщика. Вторая - ориентирована на получение рабочего, не содержащего отладочного кода EXE-файла. В любой из них можно изменить настройки компилятора и компоновщика. Используя цепочку Build - Configurations можно добавить или удалить модификацию. Каждая из модификаций может быть сгенерирована в своей директории. По умолчанию имя директории для модификации совпадает с именем модификации, но оно может быть изменено в результате выполнения цепочки Build - Settings - General - установить директории в полях области Output directories - OK. Генерируемую по умолчанию модификацию можно изменить, выполнив цепочку Build - Set (Active) Default Configuration - выбор конфигурации - OK.

В проекте используются файлы с исходными текстами программ и файлы ресурсов, хранящие, например, диалоговые окна. Также могут быть ссылки на модули (оператор USE), файлы с объектным кодом (ссылка указывается как параметр опции построителя) и динамически подключаемые библиотеки (в результате употребления атрибута DLLIMPORT). Кроме того, в исходном коде можно указать путь к библиотекам, содержащим вызываемые процедуры, применив нужное число раз директиву $objcomment lib : "*имя библиотеки*". Указанное имя будет сохранено в объектном коде и затем использовано компоновщиком при сборке приложения.

Модули, если их исходный код не содержится в проекте, ищутся компилятором по имени, указанному в операторе USE. Расширение искомых файлов - MOD. Порядок поиска:

1) поиск в директории, содержащей файлы проекта;
2) в директориях, заданных в опции компилятора /I в порядке их следования. При задании в опции более одной директории имена директорий разделяются пробелами: /I "myfiles/" /I "mylib/";
3) в директориях, заданных в переменной окружения INCLUDE, например в файле autoexec.bat.

Кроме того, в проект можно включить файл с исходным текстом, употребив строку INCLUDE или директиву $INCLUDE. Строка (директива) может содержать полное имя включаемого файла (т. е. имя и путь к файлу). Если же полное имя файла не указано, то компилятор ищет файл в той же последовательности, какая применяется для модуля.

При нахождении модуля (включаемого файла) поиск прекращается. В случае неудачи генерируется сообщение об ошибке.

В среде DS директории для поиска модулей и включаемых файлов можно установить, выполнив цепочку Tools - Options - Show directories for Include files - добавить нужную директорию - OK. Также директории указываются и в результате выполнения цепочки (Project) Build - Settings – Fortran - Preprocessors - занести в поле INCLUDE and USE paths пути к файлам - OK. Путь к файлам завершается слешем. При наличии нескольких путей они разделяются запятыми, например: myfiles/, mylib/. Заданные опции компилятора отображаются в поле Project Options.

8.4. Главная программа

Создаваемый в результате компиляции и компоновки исполняемый файл называется *приложением*. Любое приложение содержит одну главную программу, которая в общем случае имеет вид:

[PROGRAM *имя программы*]
[*операторы описания*]
[*исполняемые операторы*]
[CONTAINS
внутренние процедуры]
END [PROGRAM [*имя программы*]]

Оператор PROGRAM необязательный. Однако же если он присутствует, то должно быть задано и *имя программы* - любое правильно сформированное имя Фортрана. Если оператор END содержит *имя программы*, то оно должно совпадать с именем, заданным в операторе PROGRAM.

Главная программа не может содержать операторы MODULE и BLOCK DATA. Раздел описаний не может содержать операторы и атрибуты OPTIONAL, INTENT, PUBLIC и PRIVATE. Включение атрибута или оператора SAVE не имеет никакого эффекта. Операторы SUBROUTINE, FUNCTION, RETURN и ENTRY не могут появляться среди исполняемых операторов главной программы, но могут размещаться после оператора CONTAINS.

Выполнение приложения всегда начинается с первого исполняемого оператора главной программы. Оператор END, если в результате вычислений на него передается управление, завершает выполнение приложения. Оператор может иметь метку, используя которую можно перейти на END от других исполняемых операторов. Нормальное завершение приложения может быть также выполнено оператором STOP, который может появляться как в главной программе, так и в процедуре.

8.5. Внешние процедуры

В Фортране могут быть определены два типа процедур: *подпрограммы* и *функции*. *Функция* отличается от *подпрограммы* тем, что вызывается из выражения и возвращает результат, который затем используется в этом выражении. Тип возвращаемого результата определяет тип функции. При задании *внешней* функции следует объявлять ее тип в разделе объявлений вызывающей программной единицы так же, как это делается для других объектов данных.

Процедуру следует оформлять в виде *функции*, если ее результат можно записать в одну переменную, в противном случае следует применить *подпрограмму*.

Структура подпрограммы имеет вид:

заголовок подпрограммы
[*операторы описания*]
[*исполняемые операторы*]
[CONTAINS
внутренние процедуры]
END [SUBROUTINE [*имя подпрограммы*]]

Аналогично выглядит структура функции:

заголовок функции
[*операторы описания*]
[*исполняемые операторы*]
[CONTAINS
внутренние процедуры]
END [FUNCTION [*имя функции*]]

Функция содержит результирующую переменную, в которую устанавливается возвращаемый функцией результат. В Фортране 77 имя результирующей переменной всегда совпадало с именем функции. Ныне в предложении RESULT можно задать для результата другое имя.

Оператор CONTAINS в процедуре играет такую же роль, что и в главной программе. Выполнение оператора END приводит к передаче управления в вызывающую программную единицу. Также выход из процедуры осуществляет оператор RETURN.

Заголовок подпрограммы содержит оператор SUBROUTINE, а заголовок функции - оператор FUNCTION. Вызов подпрограммы выполняется оператором

CALL *имя подпрограммы*([*список фактических параметров*])

Вызов функции выполняется из выражения, например:

result = имя функции([*список фактических параметров*])

8.6. Внутренние процедуры

Внутри главной программы, внешней и модульной процедуры можно после оператора CONTAINS задать внутренние процедуры. Они имеют вид:

заголовок подпрограммы
[*операторы описания*]
[*исполняемые операторы*]
END SUBROUTINE [*имя подпрограммы*]

заголовок функции
[*операторы описания*]
[*исполняемые операторы*]
END FUNCTION [*имя функции*]

В отличие от внешних и модульных процедур внутренние процедуры не могут содержать других внутренних процедур. В отличие от внешних процедур внутренние процедуры, так же как и модульные, обязательно содержат в операторе END слово FUNCTION в случае функции или SUBROUTINE в случае подпрограммы. Внутренняя процедура имеет доступ к объектам носителя, включая возможность вызова *других* его внутренних процедур. Внутренние процедуры обладают явно заданным интерфейсом (разд. 8.11.3), поэтому тип внутренней функции не должен объявляться в ее носителе.

8.7. Модули

Модуль используется для задания глобальных данных и модульных процедур. Он имеет вид:

MODULE *имя модуля*
[*раздел описаний*]
[CONTAINS
модульные процедуры]
END [MODULE [*имя модуля*]]

Раздел описаний может содержать определения встроенных типов данных, объявления данных, функций и их атрибутов, интерфейсные блоки и *namelist*-группы. При объявлении данных им могут быть присвоены начальные значения. *Операторы объявления* данных могут содержать атрибуты ALLOCATABLE, AUNOMATIC, DIMENSION(*dim*), EXTERNAL, INTENT, INTRINSIC, OPTIONAL, PARAMETER, POINTER, PRIVATE, PUBLIC, SAVE, STATIC, TARGET и VOLATILE (последний атрибут применим только в CVF). Каждый атрибут может быть задан в операторной форме. Также *раздел описаний* может включать операторы COMMON, DATA, EQUIVALENCE, IMPLICIT, NAMELIST, USE. Раздел описаний не может содержать ни одного выполняемого оператора.

Следующие за оператором CONTAINS модульные процедуры должны завершаться END SUBROUTINE [*имя подпрограммы*] или END FUNCTION [*имя функции*]. Модульные процедуры, в свою очередь, после оператора CONTAINS могут содержать внутренние процедуры. *Имя модуля*, если оно следует за END MODULE, должно совпадать с именем в заголовке модуля.

Доступ к модулю в программной единице осуществляется посредством оператора USE, который предваряет раздел объявлений программной единицы.

Объявленные в операторах описания модуля имена доступны:

- в модульных процедурах;
- во внутренних процедурах модуля;
- в использующих модуль программных единицах (если имена не объявлены PRIVATE).

Модульные процедуры доступны:

- в других модульных процедурах модуля;
- в использующей модуль программной единице (если они не объявлены PRIVATE).

Внутренняя процедура модуля доступна только в содержащей ее модульной процедуре - носителе.

Объявленные в раздел описаний модуля переменные и модульные процедур, не защищенные атрибутом PRIVATE, называются *глобальными объектами модуля*.

На модуль, используя оператор USE, можно сослаться:

- в главной программе;
- во внешней процедуре;
- в другом модуле.

При ссылке на модуль в другом модуле следует следить, чтобы модуль не ссылался сам на себя ни прямо, ни косвенно через другие модули.

Замечание. Одно из применений модулей - это накопление интерфейсов внешних процедур, в том числе написанных на иных языках программирования. Например, поставляемый с CVF модуль DFCOM содержит интерфейсы, применяемые с объектами ActiveX процедур. Сами же процедуры содержатся в библиотеке dfcom.lib.

Пример:

```
module testmod
 ! Переменные a, b и c доступны в smod и smod2 и в любой программной единице,
 ! содержащей ссылку USE TESTMOD
 integer, save :: a = 1, b = 1
 integer, private :: c = 1          ! Объявленная PRIVATE переменная c
contains                            ! доступна только в модуле TESTMOD
```

```
  subroutine smod(d)
    integer d, c2 /1/
    d = 2
    b = 1 + c
    call smod2(d)
    print *, 'smod  :', d, b
  contains
    subroutine smod2(d)          ! Подпрограмма smod2 доступна
      integer d                  ! только подпрограмме smod
      print *, 'smod2 :', d, b
      d = d + 1
      b = b + c2                 ! c2 доступна в smod2 благодаря ассоциированию
    end subroutine smod2         ! через носитель
  end subroutine smod
end module testmod

program prom
 use testmod                     ! Ссылка на модуль TESTMOD
 print *, 'prom_1:', a, b        ! a и b доступны в prom и osub благодаря
 call smod(a)                    ! use-ассоциированию
 print *, 'prom_2:', a, b
 call osub( )
 print *, 'prom_3:', a, b
end program prom

subroutine osub( )
 use testmod
 print *, 'osub  :', a, b
 a = 4
 b = 4
end subroutine osub
```

Результат:

```
prom_1:     1       1
smod2 :     2       2
smod  :     3       3
prom_2:     3       3
osub  :     3       3
prom_3:     4       4
```

Объявленные в модуле TESTMOD переменные *a* и *b* являются глобальными в том смысле, что они видны в любом ссылающемся на модуль программном компоненте. Такой способ передачи данных называется *use-ассоциированием*.

Также значения переменных *a*, *b* и *c*, объявленные в модуле TESTMOD, известны модульной подпрограмме *smod* и ее внутренней подпрограмме *smod*2. Такой способ передачи данных называется *ассоциированием через носитель*. Благодаря ассоциированию через носитель внутренняя

подпрограмма *smod*2, носителем которой является модульная подпрограмма *smod*, имеет доступ к локальной переменной *c*2 подпрограммы *smod*.

Кроме того, в примере главная программа *prom*, модульная подпрограмма *smod*, внутренняя подпрограмма *smod*2 обмениваются данными через параметр *d*. Такая передача данных называется *ассоциированием параметров*.

Рассмотренное *use*-ассоциирование позволяет передавать не только данные, но и статус размещаемых массивов (разд. 4.8.2) и ссылок. Например:

```
module mod
 integer, pointer :: a(:)
end module

program tpo
use mod
integer, parameter :: m = 4, n = 5
integer, target :: b(5) = 5
a => b                          ! Прикрепляем ссылку к адресату b
call shost( )                   ! Статус ссылки будет известен в shost
print *, a                      !       3       3       3
end program

subroutine shost( )
 use mod
 integer, target :: c(3) = 3
 print *, associated(a)         ! T
 print *, a                     !       5       5       5       5       5
 a => c
end subroutine shost
```

Исходный текст модуля может быть размещен в том же файле, в котором размещены и использующие его программные единицы. При этом его текст должен предшествовать ссылкам на модуль. Можно разместить модуль в отдельном файле, например модуль TESTMOD (один или вместе с другими модулями) может быть размещен в файле mofil.f90. При компиляции этого файла (файл должен быть включен в проект) для каждого его модуля будет создан файл, имеющий имя модуля и расширение MOD. Так, для модуля TESTMOD будет создан файл testmod.mod. При работе с откомпилированными файлами модулей присутствие исходных текстов модулей в проекте необязательно, но компилятору должны быть известны пути к откомпилированным файлам или к содержащим эти файлы библиотекам.

8.8. Оператор USE

Доступ к модулю выполняется посредством использования оператора USE. Если, например, задан оператор

USE TESTMOD

то программная единица получает доступ ко всем не имеющим атрибута PRIVATE объявленным в разделе описания модуля объектам данных и модульным процедурам модуля. Причем все объекты модуля известны в использующей его программной единице под теми именами, которые они имеют в модуле.

В то же время оператор USE позволяет:

- ограничить доступ к объектам модуля за счет применения параметра ONLY;
- использовать в программной единице для объектов модуля другие имена.

Например, на модуль TESTMOD (разд. 8.7) можно сослаться так:

```
use testmod, only : a, smod
```

или так:

```
program prom
 use testmod, va => a, prosub => smod
 print *, 'prom_1:', va, b
 call prosub(va)
 call osub( )
end program prom
```

В первом случае в использующей модуль TESTMOD программной единице будут видны только переменная *a* модуля и его подпрограмма *smod*. Во втором случае в результате переименования переменная *a* модуля TESTMOD будет доступна в *prom* под именем *va*, а модульная подпрограмма *smod* - под именем *prosub*. Глобальная переменная *b* модуля будет доступна под ее собственным именем. Механизм взаимодействия модуля и использующей его программной единицы, разумеется, сохранится.

Переименование и ограничение доступа посредством ONLY применяется в основном для предотвращения конфликта имен. Например, если программная единица использует два модуля, в которых есть одноименные глобальные объекты, то для предотвращения конфликта по крайней мере один из этих объектов следует использовать под другим именем. Если же, например, имя глобального объекта модуля конфликтует с именем локального объекта использующей модуль программной единицы и к тому же этот объект модуля в этой программной единице не применяется, то конфликт можно преодолеть, ограничив опцией ONLY доступ к этому объекту.

Дадим теперь общее представление двух форм оператора USE:

USE *имя модуля* [, *список переименований*]

USE *имя модуля*, ONLY: [*only-список*]

Список переименований содержит переименования глобальных объектов модуля. Каждый элемент списка имеет вид

local-name => use-name

и означает, что объект модуля с именем *use-name* будет доступен в использующей модуль программной единице под именем *local-name*. В общем случае можно для одного *use-name* использовать несколько разных *local-name*, например:

```
use testmod, prosub => smod, prosub2 => smod
```

only-список ограничивает доступ к глобальным объектам модуля. Элементом списка может быть любое глобальное имя, в том числе задаваемый оператор, задаваемое присваивание, родовое имя, имя объявленного в модуле производного типа данных. Например:

```
module mymod
 type point                    ! Объявляем в модуле производный тип данных point
  real(4) :: x, y, z
 end type point
end module mymod

program t2
 use mymod                     ! Теперь тип данных point доступен в t2
 type(point) :: pt             ! Объявляем переменную pt типа point
 pt = point(1.0, 2.0, 3.0)     ! Даем переменную pt типа point начальные значения
 print *, pt                   !       1.000000       2.000000       3.000000
end program t2
```

Также *only-список* может включать элемент вида

[*local name =>*] *use-name*

Находящиеся в *only-списке* объекты модуля не могут иметь атрибут PRIVATE. Если опция ONLY задана, то в использующей модуль программной единице доступны только размещенные в *only-списке* объекты модуля.

Программная единица может содержать более одного оператора USE для любого модуля, например:

```
use testmod, only : va => a, b
use testmod, only : prosub => smod
```

В этом случае *only-списки* сцепляются в один *only-список*. Если же хотя бы один оператор USE использован без опции ONLY, то использующей модуль программной единице будут доступны все глобальные объекты модуля, а присутствующие переименования в *списках переименований* и *only-списках* сцепляются в единый *список переименований*.

Пример:

```
module mymod
 real :: a = 5.0
end module mymod
```

```
program t3
 use mymod, a2 =>a
 ! Теперь переменная a модуля MYMOD доступна в t3 под именем и a2
 print *, a, a2                    !      0.000000      5.000000
 ! Переменную a можно использовать в t3 как локальную
end program t3
```

Все используемые при переименовании локальные имена должны отличаться друг от друга и от локальных имен использующей модуль программной единицы. Локальные имена программной единицы должны отличаться от глобальных имен модулей, на которые в ней имеются ссылки. Например, следующий код:

```
module mymod
 real :: a = 5.0
contains
 subroutine b(d)
  real d
  d = 4.5
 end subroutine b
end module mymod
program t4
 use mymod
 b = 1.2                           ! Глобальное имя b уже использовано в MYMOD
end program t4                     ! как имя подпрограммы
```

вызовет ошибку компиляции: Error: This name has already been used as an external subroutine name. [B] b = 1.2. Если же в *t*4 добавить объявление

```
real(4) b
```

то дополнительно возникнет ошибка: Error: The attributes of this name conflict with those made accessible by a USE statement. [B] real(4) b.

Избежать ошибки можно, во-первых, удалив из *t*4 ссылку *use mymod*, во-вторых, выполнив переименование, например *use mymod*, *b*2 => *b*, и, в-третьих, изменив имя локальной переменной *b* главной программы *t*4, например на *b*3.

Один и тот же глобальный объект модуля может быть доступен под несколькими локальными именами. Это достигается либо за счет его неоднократного использования в списке переименований, либо, например, так:

```
module a
 real s, t
 ...
end module a
module b
 use a, bs => s
```

```
...
end module b

subroutine c                ! Переменная s модуля a доступна подпрограмме c
 use a                      ! под своим настоящим именем s и именем bs
 use b
...
end subroutine c
```

Имена глобальных объектов используемых программной единицей модулей могут дублироваться, если:

- два или более родовых интерфейса, доступных программной единице, имеют одно и то же имя, задают одну и ту же операцию или задают присваивание. В этом случае компилятор рассматривает все родовые интерфейсы как один;
- глобальные объекты, не являющиеся родовыми интерфейсами, доступны программной единице, но в ней не используются.

Если операторы USE содержатся в модуле, то все выбранные объекты рассматриваются как объекты самого модуля. Им можно дать атрибуты PRIVATE или PUBLIC как явно, так и по умолчанию. Задавать какие-либо иные атрибуты у выбранных объектов нельзя, но их можно включать в одну или более *namelist*-групп. Однако нельзя задать атрибут PUBLIC объекту включаемого модуля, если в последнем объект имеет атрибут PRIVATE.

Пример. Выполняется переименование имен производных типов данных.

```
module geometry             ! Определения производных типов данных
 type square
   real side
   integer border
 end type
 type circle
   real radius
   integer border
 end type
end module

program test
! Переименуем имена типов данных модуля  для локального использования
use geometry, lsquare => square, lcircle => circle
type(lsquare) s1, s2        ! Используем новые имена при объявлении
type(lcircle) c1, c2, c3    ! переменных
```

8.9. Атрибуты PUBLIC и PRIVATE

Атрибуты PUBLIC и PRIVATE могут быть даны только объектам модуля. Атрибут PUBLIC указывает, что объект модуля может быть доступен в результате *use*-ассоциирования в использующих модуль

программных единицах. Напротив, если объект модуля имеет атрибут PRIVATE, то он может быть использован только внутри модуля. Задание атрибутов может быть выполнено как отдельным оператором, так и при объявлении типа:

PUBLIC | PRIVATE [[::] *объекты модуля*]

type-spec, PUBLIC | PRIVATE [, *атрибуты*] :: *объекты модуля*

Объекты модуля могут включать имена переменных, констант, процедур, *namelist*-групп, производных типов и родовых описаний.

type-spec - оператор объявления встроенного или производного типа данных.

По умолчанию объекты модуля имеют атрибут PUBLIC. Если оператор PRIVATE задан без списка объектов модуля и нет объектов, для которых явно задан атрибут PUBLIC, то действие атрибута PRIVATE распространяется на все объекты модуля, даже если они объявлены до оператора PRIVATE. Например:

```
module pupr                    ! Переменные a и b имеют атрибут PRIVATE
 real a
 private
 integer b
```

Аналогичный эффект вызывает задание без списка объектов модуля оператора PUBLIC. В модуле может быть только один оператор без списка объектов модуля (PUBLIC или PRIVATE).

Объекту не может быть дан атрибут PUBLIC, если он уже имеет атрибут PRIVATE.

Родовое описание, если оно не имеет атрибут PRIVATE, является PUBLIC, даже если одно или все его специфические имена объявлены PRIVATE.

Если *namelist*-группа имеет атрибут PUBLIC, то ни один из ее компонентов не может иметь атрибут PRIVATE.

Компоненты объявленного PUBLIC производного типа имеют атрибут PUBLIC, за исключением компонентов, которые имеют атрибут PRIVATE.

Пример:

```
module pupr
 type pri                      ! Все переменные типа pri будут
  private                      ! иметь атрибут PRIVATE
  integer ix, iy
 end type pri
 type, public :: pub
  real x, y
  type(pri) epin               ! Этот компонент типа pub имеет
 end type pub                  ! атрибут PRIVATE
 type(pub), public :: ep = pub(3.0, 4.0, pri(3, 3))
```

```
 real, public :: a = 3.0, b = 4.0
 public :: length
 private :: square                ! Подпрограмма square доступна
contains                          ! только в модуле pupr
  real function length(x, y)
   real, intent(inout) :: x, y
   call square(x, y)
   length = sqrt(x + y)
  end function
  subroutine square(x1, y1)
   real, intent(inout) :: x1, y1
   x1 = x1 * x1
   y1 = y1 * y1
 end subroutine
end module pupr
program gopu
 use pupr
 print *, length(ep%x, ep%y)     !     5.000000
 print *, length(a, b)           !     5.000000
end
```

8.10. Операторы заголовка процедур

Полный синтаксис оператора заголовка подпрограммы:

[RECURSIVE] SUBROUTINE *имя подпрограммы* &
    [([*список формальных параметров*])]

Общий вид оператора заголовка функции:

[*type*] [RECURSIVE] FUNCTION *имя функции* &
    ([*список формальных параметров*]) [RESULT (*имя результата*)]

8.10.1. Общие характеристики операторов заголовка процедур

Имя процедуры (*подпрограммы* и *функции*) может быть глобальным и внешним или внутренним в процедуре-носителе. *Имя процедуры* не может появляться в операторах AUTOMATIC, COMMON, EQUIVALENCE, DATA, INTRINSIC, NAMELIST, SAVE. *Имя подпрограммы* не может появляться в операторах объявления типа.

Список формальных параметров может содержать имена переменных и формальных процедур. В случае подпрограммы формальным параметром может быть и обозначающая альтернативный возврат звездочка. *Формальные параметры* могут отсутствовать, если передача данных выполняется посредством *use*-ассоциирования, ассоциирования через носитель или *common*-блоки.

Процедура может содержать любые операторы, кроме BLOCK DATA и PROGRAM. До оператора CONTAINS процедура не может содержать

операторы FUNCTION и SUBROUTINE. Внутренние процедуры не могут содержать операторы ENTRY и CONTAINS и другую внутреннюю процедуру. Внутренние процедуры размещаются между операторами CONTAINS и END главной программы или процедуры-носителя.

Если процедура внешняя, то ее имя является глобальным и не должно совпадать с другим глобальным именем, а также не должно быть использовано для локального имени в вызывающей программной единице. В случае внутренней процедуры ее имя является локальным и область его действия ограничена носителем.

Если формальный параметр имеет атрибут OPTIONAL, то при вызове соответствующий фактический параметр может быть опущен. Типы формальных параметров могут быть заданы внутри процедуры как неявно, так и явно (последнее предпочтительнее). Имена формальных параметров не могут появляться в операторах AUTOMATIC, COMMON, DATA, EQUIVALENCE, INTRINSIC, SAVE или STATIC.

При вызове процедуры передаваемые фактические параметры должны быть согласованы с соответствующими формальными по порядку (за исключением случаев, когда используются вызовы с ключевыми словами), по числу (за исключением случаев, когда заданы атрибуты OPTIONAL или C), по типу и разновидности типа. Компилятор проверяет соответствие параметров. При обнаружении несоответствия, как правило, генерируются ошибки. Полная проверка соответствия фактических и формальных параметров выполняется компилятором при явном задании интерфейса к процедуре. Явным интерфейсом обладают модульные и внутренние процедуры.

Полезно явно задавать интерфейс и к внешним процедурам, а в большом числе случаев он просто необходим (разд. 8.11.3).

Если вызываемая процедура находится в динамической библиотеке (DLL), то необходимо задавать ее интерфейс и использовать с ней атрибуты DLLEXPORT или DLLIMPORT [1].

Выход из процедуры осуществляется в результате выполнения либо оператора END, либо оператора RETURN. Последний может быть размещен где угодно среди исполняемых операторов процедуры.

8.10.2. Результирующая переменная функции

Функция обязана содержать результирующую переменную, в которую помещается возвращаемый функцией результат.

Имя результирующей переменной задается предложением RESULT или совпадает с именем функции, если это предложение опущено. Задаваемое предложением RESULT *имя результата* не может совпадать с *именем функции*.

Тип результирующей переменной определяет тип функции и может быть задан посредством указания *type* в заголовке функции.

type - объявление типа и разновидности типа результирующей переменной (возвращаемого функцией результата). Может быть любого встроенного типа.

type может быть опущен. Тогда тип результирующей переменной может быть задан явно в одном из операторов объявления функции, в операторе IMPLICIT или неявно. Последнее невозможно, если заданы оператор IMPLICIT NONE или директива $DECLARE.

Если в операторе заголовка задан *type*, то имя результирующей переменной не должно появляться в разделе объявлений функции.

Примеры объявления результирующей переменной:

```
function imax(a, n)                ! Результирующая переменная imax;
 integer a(n)                      ! ее тип INTEGER задан неявно
 ...                               ! Исполняемые операторы
logical function flag(a, n)        ! Результирующая переменная flag;
 integer a(n)                      ! ее тип LOGICAL задан явно

function flag(a, n)                ! Результирующая переменная flag;
 logical flag                      ! ее тип LOGICAL задан явно

function flag(a, n) result(vf)     ! Результирующая переменная vf;
 logical vf                        ! ее тип LOGICAL задан явно
```

Если тип внешней функции определен без учета правил умолчаний о типах данных или заданы оператор IMPLICIT NONE или директива $DECLARE, то тип функции либо должен быть объявлен в вызывающей программной единице, либо там должен быть задан интерфейс к этой функции. На модульные и внутренние функции это требование не распространяется, поскольку они и без того имеют явно заданный интерфейс.

Пример:

```
logical function flag(a, n) result (vf)
 ...
 vf = ...                          ! Определяем значение результирующей переменной
end

program fude
 logical flag, fl                  ! Объявляем функцию flag в вызывающей программе
 ...                               ! Тип объявляемой функции определяется типом
 fl = flag(a, n)                   ! результирующей переменной
 ...
end
```

Если результатом функции является переменная производного типа, массив или ссылка, то *type* опускается и результирующая переменная объявляется в разделе объявлений функции.

Результирующая переменная подобна параметру с видом связи OUT. При входе в функцию она не определена, далее получает значение, которое затем используется в вызывающей программной единице.

Результирующая переменная может использоваться в выражениях функции, и в результате вычислений она должна получить значение. Последнее значение результирующей переменной используется в выражении, из которого функция вызвана. Результирующая переменная после выполнения функции может быть не определена, если она является ссылкой и вызов функции выполнен не из выражения (разд. 3.11.8). Обычно результирующая переменная получает значение в результате присваивания. Но это необязательно. Например, в случае символьной функции результирующая переменная может быть определена после выполнения оператора (разд. 3.8.7)

WRITE(*имя результата*, *спецификатор формата*) *выражение*

Результирующая переменная может быть скаляром или массивом любого встроенного и производного типа, также она может быть и ссылкой (разд. 3.11.8). Результатом функции не может быть перенимающий размер массив.

Так же как и в случае подпрограммы, функция может возвращать данные и через передаваемые по ссылке параметры (параметры с видом связи OUT или INOUT). Однако такой способ передачи данных может привести к побочным эффектам (разд. 8.11.6) и поэтому не может быть рекомендован.

Пример. Найти сумму последних отрицательных элементов массивов *a* и *b*. Поиск последнего отрицательного элемента в массиве выполним в функции *finel*.

```
program nel
 integer :: a(8) = (/1, -1, 2, -2, 3, -3, 4, -4 /)
 integer :: b(10) = (/1, 2, 3, 4, 5, -1, -2, -3, -4, -5 /)
 integer :: finel                    ! Объявляем тип функции
 print *, finel(a, 8) + finel(b, 10) ! Вызов функции из выражения
end                                  !        -9

function finel(c, n)
 integer finel                       ! Объявляем тип результирующей переменной
 integer c(n), i                     ! Используем массив заданной формы
 finel = 0                           ! Определим результирующую переменную на
 do i = n, 1, -1                     ! случай, если в массиве нет отрицательных
  if(c(i) < 0) then                  ! элементов
   finel = c(i)
   return                            ! Возвратим последний отрицательный элемент
  end if
 end do
end function
```

8.11. Параметры процедур

Обмен данными между процедурой и вызывающей программной единицей может быть выполнен через параметры процедуры.

Параметры, используемые при вызове процедуры, называются *фактическими*.

Параметры, используемые в процедуре, называются *формальными*.

Пример. Сформировать вектор *c*1 из элементов вектора *a*, которых нет в векторе *b*1. Затем сформировать вектор *c*2 из элементов вектора *a*, которых нет в векторе *b*2. Формирование массивов выполним в подпрограмме *fobc*.

```
program part
integer, parameter :: na = 10, nb1 = 5, nb2 = 7
integer :: a(na) = (/ 1, -1, 2, -2, 3, -3, 4, -4, 5, -5 /)
integer :: b1(nb1) = (/ 1, -1, 2, -2, 3 /)
integer :: b2(nb2) = (/ 1, -1, 2, -2, 3, -3, 4 /)
integer c1(na), c2(na), nc1, nc2
call fobc(a, na, b1, nb1, c1, nc1)      ! Формируем массив c1
call fobc(a, na, b2, nb2, c2, nc2)      ! Формируем массив c2
write(*, *) c1(:nc1)                    !      -3      4      -4      5      -5
write(*, *) c2(:nc2)                    !      -4      5      -5
end program

subroutine fobc(a, na, b, nb, c, nc)
 integer na, nb, a(na), b(nb)           ! Входные формальные параметры
 integer c(na), nc                      ! Выходные формальные параметры
 integer i, j, va
 nc = 0                                 ! Число элементов в формируемом массиве
 loop_a: do i = 1, na                   ! Имя DO-конструкции использует
  va = a(i)                             ! оператор CYCLE loop_a
  do j = 1, nb
   if( va == b(j) ) cycle loop_a
  end do
  nc = nc + 1
  c(nc) = va
 end do loop_a
end subroutine fobc
```

В операторе CALL *fobc* присутствуют *фактические параметры*. Тогда как присутствующие в операторе SUBROUTINE *fobc* параметры *a*, *na*, *bc* и *m* являются *формальными*.

8.11.1. Соответствие фактических и формальных параметров

При вызове процедуры между фактическими и формальными параметрами устанавливается соответствие (формальные параметры ассоциируются с соответствующими фактическими). Так, в нашем примере при первом вызове подпрограммы *fobc* фактическому параметру *a*

соответствует формальный параметр *a*, фактическому параметру *b*1 - формальный параметр *b* и т. д. Типы соответствующих параметров совпадают. Как видно из примера, имена соответствующих фактических и формальных параметров могут различаться.

В нашем примере скорее всего фактический и соответствующий ему формальный параметр будут адресовать одну и ту же область памяти. Правда, это справедливо не во всех случаях. Так, если фактическим параметром является сечение массива, то при вызове процедуры компилятор создаст его копию, которую и будет адресовать формальный параметр. При выходе из процедуры (если параметр имеет вид связи OUT или INOUT) произойдет обратная передача данных из копии в сечение-параметр.

Фактическими параметрами могут быть выражения, в том числе буквальные и именованные константы, простые переменные, массивы и их сечения, элементы массивов, записи, элементы записей, строки и подстроки, а также процедуры и встроенные функции; в случае подпрограммы именем фактического параметра может быть и метка.

Формальными параметрами могут быть переменные (полные объекты), процедуры и звездочка (*).

Фактические и формальные параметры могут иметь атрибуты, например POINTER или TARGET.

Устанавливая соответствие между фактическими и формальными параметрами, следует придерживаться приведенных в табл. 8.2 правил.

Таблица 8.2. Фактические и формальные параметры

Фактический параметр	*Формальный параметр*
Скалярное выражение	Скалярная переменная
Нескалярное выражение (массив, сечение массива...)	Массив
Процедура	Процедура
*Метка (только для подпрограмм)	* (звездочка)

Замечания:

1. Если формальный параметр является ссылкой, то и соответствующий фактический параметр тоже должен быть ссылкой.

2. Если фактическим параметром является строка, то формальным параметром может быть строка, перенимающая длину (разд. 3.8.2).

3. Если фактическим параметром является элемент массива, то соответствующим формальным параметром может быть массив (разд. 4.9.1).

4. Если фактическим параметром является массив, то формальным параметром может быть массив заданной формы, или перенимающий форму массив, или перенимающий размер массив (разд. 4.9).

5. Если фактический параметр является внешней процедурой, то он должен иметь атрибут EXTERNAL. Если же фактический параметр является встроенной процедурой, то он должен быть объявлен с атрибутом INTRINSIC.

8.11.2. Вид связи параметра

Формальные параметры разделяются на *входные, выходные и входные/выходные.* Входной формальный параметр получает свое значение от соответствующего фактического параметра. Выходной - передает свое значение соответствующему фактическому параметру. Входные/выходные - осуществляют связь в двух направлениях.

В подпрограмме *fobc* (разд. 8.11) формальные параметры *a*, *na*, *b* и *nb* являются входными. Параметры *c* и *nc* – выходными, т. е. их значения определяются в процедуре и потом уже используются в вызывающей программной единице. Такое разделение формальных параметров примера на входные/выходные мы выполнили, исходя из совершаемых программой действий. На самом деле вид связи формального параметра можно задать явно, использовав атрибут INTENT, например:

```
subroutine fobc(a, na, b, nb, c, nc)
 integer, intent(in) na, nb, a(na), b(nb)
 integer, intent(out)  c(na), nc
 integer i, j, va
```

Для задания атрибута INTENT может быть применен оператор INTENT:

```
subroutine fobc(a, na, b, nb, c, nc)
 integer na, nb, nc, a(na), b(nb), c(na), i, j, va
 intent(in) na, nb, a, b
 intent(out) c, nc
```

Синтаксис оператора INTENT:

INTENT (*spec*) [::] *vname*

Синтаксис атрибута INTENT:
type-spec, INTENT (spec) [, attrs] :: vname

spec - вид связи формального параметра, *spec* может принимать одно из трех значений:

- IN - формальный параметр является входным и не может быть изменен или стать неопределенным в процедуре. Ассоциированный с ним фактический параметр может быть выражением, например константой или переменной;

- OUT - формальный параметр является выходным. При входе в процедуру такой формальный параметр всегда не определен и поэтому должен получить значение до его использования. Ассоциированный с ним фактический параметр должен быть определяемым, например переменной, подстрокой или элементом записи;
- INOUT - формальный параметр может как получать данные от фактического параметра, так и передавать данные в вызывающую программную единицу. Как и в случае вида связи OUT, ассоциированный с ним фактический параметр должен быть определяемым (не должен быть, например, константой).

vname - разделенные запятыми имена формальных параметров.
type-spec - спецификация любого типа данных.
attrs - список иных атрибутов формального параметра.

Если атрибут INTENT не задан, то способ использования формального параметра определяет ассоциированный с ним фактический параметр. Так, формальный параметр не должен переопределяться в процедуре, если ассоциированный с ним фактический параметр выражение или константа.

```
real :: length, x = 3.0, y = 4.0, r
r = length(3.0, 4.0)              ! Этот вызов ошибочен
r = length(x, y)                  ! Этот вызов допустим
end

real function length(x, y)        ! Первый вызов ошибочен, поскольку
 real x, y                        ! формальные параметры x и y
 call square(x, y)                ! переопределяются в подпрограмме square
 length = sqrt(x + y)
end function

subroutine square(x1, y1)
 real, intent(inout) :: x1, y1
 x1 = x1 * x1
 y1 = y1 * y1
end subroutine
```

Если формальный параметр имеет вид связи IN, то он не должен быть использован в качестве фактического параметра, ассоциируемого с формальным параметром, вид связи которого OUT или INOUT. Так, в предыдущем примере формальные параметры *x*, *y* функции *length* не должны иметь вид связи IN.

Если функция задает перегружаемую операцию, то формальные параметры обязаны иметь вид связи IN. Если подпрограмма определяет задаваемое присваивание, то первый ее формальный параметр должен иметь вид связи OUT или INOUT, а второй - IN.

Недопустимо использование атрибута INTENT:

- для формальных параметров с атрибутом POINTER;

- для формальных параметров - процедур (формальных процедур).

8.11.3. Явные и неявные интерфейсы

Интерфейс между процедурой и вызывающей ее программной единицей считается заданным, если вызывающей программной единице известны имя процедуры, ее вид (подпрограмма или функция), свойства функции (если процедура - функция), имена, положение и свойства формальных параметров.

В Фортране 77 интерфейс к вызываемой процедуре полностью неизвестен и он устанавливается при вызове по списку фактических параметров. Устанавливаемый таким образом интерфейс называется *неявным*. Такие вызовы могут быть причиной ряда ошибок, поскольку компилятор не всегда имеет возможность, например, проверить, соответствуют ли фактические и формальные параметры так, как это им положено.

Вызовы внешних процедур с неявным интерфейсом допустимы и в современном Фортране. В случае функции при неявном интерфейсе ее тип и разновидность типа задаются в одном из операторов объявления типа вызывающей программной единицы или устанавливаются в соответствии с действующими правилами умолчания.

Однако формальные параметры процедур и процедуры-функции могут обладать дополнительными свойствами, о которых должен знать компилятор, чтобы правильно организовать доступ к коду процедуры. Чтобы сообщить компилятору такие сведения, между вызывающей программной единицей и процедурой должен существовать *явный интерфейс*.

В случае внутренней процедуры вызывающая программная единица и ее процедура компилируются как единое целое, поэтому компилятор знает все о любой внутренней процедуре, т. е. между внутренней процедурой и вызывающей программной единицей существует явный интерфейс.

Модульная процедура вызывается либо в самом модуле, либо из программной единицы, где вызову предшествует оператор USE для этого модуля. Поэтому в обоих случаях компилятор знает все о вызываемой процедуре и, следовательно, ее интерфейс является явным.

Также все встроенные процедуры заведомо обладают явным интерфейсом.

В случае внешней процедуры к ней также может быть установлен явный интерфейс. Это делается при помощи интерфейсного блока, имеющего вид:

```
INTERFACE
  тело интерфейса
END INTERFACE
```

Тело интерфейса содержит описание одного и более интерфейсов процедур. Как правило, интерфейс процедуры - это точная копия заголовка

процедуры, объявлений ее формальных параметров, типа функции в случае процедуры-функции и оператора END процедуры. Однако в интерфейсном блоке:

- имена параметров могут отличаться от имен соответствующих формальных параметров процедуры;
- могут быть добавлены дополнительные спецификации (например, объявления локальных переменных) за исключением описаний внутренних процедур и операторов DATA и FORMAT;
- можно представлять ту же информацию при помощи другой комбинации операторов объявления.

Пример:

```
subroutine sub1(i1, i2, r1, r2)
 integer :: i1, i2
 real :: r1, r2                    ! В разделе объявлений процедуры атрибут
 ...                               ! OPTIONAL может быть опущен. Достаточно того,
                                   ! что он задан в интерфейсном блоке
end subroutine sub1

program idem
interface                          ! Интерфейс подпрограммы sub1
 subroutine sub1(int1, int2, real1, real2)
  integer :: int1, int2
  real, optional :: real1, real2
 end subroutine sub1
end interface
```

Также может быть задан интерфейс внешних процедур, написанных на других языках программирования, например на ассемблере или СИ. Наличие таких интерфейсов позволяет создавать разноязычные приложения [1].

Задание интерфейса означает, что упоминаемые в нем процедуры рассматриваются как внешние. Любая встроенная процедура с таким же именем становится недоступной. Такой же эффект имеет и упоминание имени процедуры в операторе EXTERNAL. Одновременное упоминание имени процедуры в теле интерфейса и операторе EXTERNAL недопустимо.

Интерфейсный блок размещается среди операторов описания. Удобнее всего собрать интерфейсные блоки в одном или нескольких модулях и подключать их по мере необходимости при помощи оператора USE.

Задание явного интерфейса необходимо, если:

- процедура имеет необязательные формальные параметры;
- результатом процедуры-функции является массив (разд. 4.11);
- формальным параметром процедуры является перенимающий форму массив (разд. 4.9.2), ссылка или ее адресат (разд. 3.11.8);

- длина результата символьной функции не является константой и не перенимается из вызывающей программной единицы;
- результатом процедуры-функции является ссылка;
- процедура является динамической библиотекой.

Также интерфейсные блоки употребляются:

- при вызове процедуры с ключевыми словами (разд. 8.11.4);
- при использовании задаваемого присваивания (разд. 8.12.2);
- при использовании задаваемых операций (разд. 8.12.2);
- при использовании в вызове родового имени (разд. 8.12);
- для доступа к внешним, написанным на других языках процедурам [1].

8.11.4. Ключевые и необязательные параметры

При некоторых вызовах процедуры часть фактических параметров может не использоваться. Примером такой процедуры может послужить встроенная функция SUM(*array* [, *dim*] [, *mask*]) (разд. 4.12.1), имеющая два необязательных параметра *dim* и *mask*. В этом случае соответствующие формальные параметры должны быть объявлены с атрибутом OPTIONAL. Как и другие атрибуты, OPTIONAL может быть использован в операторе объявления типа и как самостоятельный оператор.

Пример. Создать функцию *npe*(*array*, *me*, *sig*), возвращающую:

- сумму *me* первых положительных элементов массива *array*, если *sig* > 0;
- сумму *me* первых отрицательных элементов массива *array*, если *sig* < 0;
- сумму *me* первых элементов массива *array*, если *sig* = 0 или отсутствует;
- сумму всех заданных посредством *sig*-знака элементов массива, если отсутствует параметр *me*.

```
program tesop
 integer, parameter :: m = 3, n = 10
 integer :: a(n) = (/ 1, -1, 2, -2, 3, -3, 4, -4, 5, -5 /)
                                  ! Необходимо явно задать интерфейс к процедуре
 interface                        ! с необязательными формальными параметрами
  integer function npe(array, me, sig)
   integer, intent(in) :: array(:)
   integer, intent(in), optional :: me, sig
  end function npe
 end interface
 print *, npe(a, m, 1)            !       6
 print *, npe(a, sig = -1)        !     -15
 print *, npe(a)                  !       0
end program tesop

integer function npe(array, me, sig)
 integer, intent(in) :: array(:)
 integer, intent(in), optional :: me, sig
 integer  mval, sval
```

```
 integer, allocatable :: temp(:)
 if(.not. present(sig)) then          ! Использовать неассоциированный
  sval = 0                            ! необязательный формальный параметр
 else                                 ! можно только в качестве аргумента
  sval = sig                          ! функции PRESENT
 end if
 if(present(me)) then
  mval = me
 else
  mval = size(array)
 end if
 select case(sval)
 case(1:)                             ! Размер temp может быть меньше mval
  allocate(temp(min(mval, count(array > 0))))
  temp = pack(array, array > 0)
 case(:-1)
  allocate(temp(min(mval, count(array < 0))))
  temp = pack(array, array < 0)
 case(0)
  allocate(temp(mval))
  temp = array(1:mval)
 endselect
 npe = sum(temp)                      ! Возвращаемый результат
 deallocate(temp)
end function npe
```

Синтаксис оператора OPTIONAL:

OPTIONAL [::] *vname*

Синтаксис атрибута OPTIONAL:

type-spec, OPTIONAL [, *attrs*] :: *vname*

type-spec - спецификация любого типа данных.
vname - разделенные запятыми имена формальных параметров.

Атрибут может быть использован только для формальных параметров процедур. Интерфейс процедуры, содержащей атрибут OPTIONAL, должен быть описан явно. Формальный параметр, имеющий атрибут OPTIONAL, может дополнительно иметь только атрибуты DIMENSION, EXTERNAL, INTENT, POINTER и TARGET.

Если необязательный формальный параметр не задан, то ему не может быть присвоено значение и его нельзя использовать в выражении. Для определения того, задан формальный параметр или нет, используется встроенная функция

PRESENT(*a*)

где *a* - необязательный формальный параметр. Функция возвращает .TRUE., если формальный параметр *a* ассоциирован с фактическим, и .FALSE. - в противном случае.

Необязательный формальный параметр может использоваться внутри процедуры в качестве фактического параметра. Если такой необязательный параметр отсутствует, то он рассматривается как отсутствующий и в процедуре следующего уровня. Отсутствующие параметры могут распространяться на любую глубину вызова. Отсутствующий параметр может появляться в качестве фактического параметра только как полный объект, а не как его подобъект.

В Фортране 77 расположение соответствующих формальных и фактических параметров в списке параметров должно совпадать, т. е. первый формальный параметр ассоциируется с первым фактическим и т. д. Если ассоциируемый формальный параметр определяется по положению фактического параметра в списке параметров, то такой фактический параметр называется *позиционным*.

В Фортране это правило может быть нарушено, если использовать при вызове процедуры параметры с ключевыми словами. *Ключевые слова* - это имена формальных параметров, присвоенные им в интерфейсном блоке

Например, допустимы вызовы функции *npe*:

```
result = npe(sig = 1, array = a, me = m)
result = npe(a, sig = 1, me = m)
result = npe(a, sig = 1)
```

В первом вызове все параметры должны предваряться ключевыми словами. Во втором и третьем - первый параметр является позиционным, поэтому он может быть задан без ключевого слова. В третьем случае не задан второй формальный параметр, поэтому для установления связи с формальным параметром *sig* необходимо использовать вызов с ключевым словом - именем формального параметра.

Однако позиционные параметры не могут появляться в списке фактических параметров после первого появления параметра с ключевым словом. Так, ошибочен вызов

```
result = npe(array = a, m, 1)
```

8.11.5. Ограничения на фактические параметры

Стандарт устанавливает два ограничения на фактические параметры:

- должны быть исключены любые действия, влияющие на значение и доступность фактического параметра в обход соответствующего формального параметра;
- если хотя бы часть фактического параметра получает значение от формального параметра, то в процедуре ссылаться на этот фактический параметр можно только через формальный параметр.

Для иллюстрации ограничений рассмотрим пример:

```
integer a(10) /10*2/, x, xx, y /2/
common /cb/ x, xx
character(10) st /'??????????'/
call rest1(x, xx, a(1:7), a(4:10))   ! Согласно ограничению 1 в rest1
call rest2(y, st(3:7))               ! нельзя менять значение a(4:7)
print *, x, xx, a(5), y, '  ', st
contains
 subroutine rest2(y2, st2)
  integer y2
  character(*) st2
  y2 = 4                             ! Верно
  y = 6                              ! Нарушено ограничение 2
  st2 = '&&&&&'                      ! Верно
  st = '##########'                  ! Нарушено ограничение 2
 end subroutine rest2
end
subroutine rest1(x2, xx2, a, a2)
 integer x2, xx2, a(*), a2(*)
 common /cb/ x, xx
 x = 1                               ! Нарушено ограничение 1
 x2 = 11                             ! Верно
 xx2 = 3                             ! Верно
 xx = 33                             ! Нарушено ограничение 1
 a(1:3) = 22                         ! Верно
 a2(5:7) = 44                        ! Верно
 a(4:7) = 5                          ! Нарушено ограничение 1
 a2(1:4) = 7                         ! Нарушено ограничение 1
end
```

Хотя пример насыщен нарушениями, CVF и FPS не выдадут ни одного сообщения или предупреждения об ошибке. Однако это совсем не означает, что программа отработает правильно. В общем случае результат непредсказуем. Это видно, в частности, из выведенных результатов:

```
CVF:      11  1107558400       7       6  ##########
FPS:      11  1107558400       5       6  ##########
```

Аналогичным образом если переменная, например *xx*, доступна процедуре *rest*1 через модуль и одновременно ассоциируется с формальным параметром *xx*2 этой процедуры, то будет нарушено ограничение 1 при попытке изменить значение *xx* в процедуре *rest*1.

8.11.6. Запрещенные побочные эффекты

Стандарт разрешает не вычислять часть выражения, если значение этого выражения может быть определено и без этого. Так, в примере

```
logical g, flo
real :: x = 5.0, y = 4.0, z = 7.0
g = x > y .or. flo(z)
print *, g, z                    ! FPS:      T          7.0000
end                              ! CVF:      T        100.0000

logical function flo(z)
 z = 100
 flo = .true.
end function flo
```

в FPS не будет выполнено обращение к логической функции *flo*. В соответствии с положениями стандарта значение переменной *z* должно после вычисления выражения стать неопределенным. Хотя в FPS переменная *z* и сохранит свое значение, а в CVF изменит, совершенно очевидно, что следует избегать подобных вызовов функции. Действительно, если бы значение *x* было равно 3.0, то после вычисления выражения в FPS мы получили бы совсем иное значение для *z* - число 100.0.

Другой пример, когда стандарт допускает неполное вычисление выражения:

```
character(len = 2) :: stre, st1 = 'd1', st2 = 'd2'
! stfun - символьная функция. Стандарт разрешает не выполнять вызов stfun,
! поскольку длина результата stre равна st1, и он полностью определяется первым
! операндом выражения st1 // stfun(st2)
stre = st1 // stfun(st2)
print *, st2

contains

 character(2) function stfun(st2)
  character(*) st2
  st2 = 'd3'
  stfun = 'd4'
 end function stfun
end
```

Заметим, однако, что и CVF и FPS обратятся к функции *stfun* и переменная *st*2 получит значение 'd3'.

Существует и другое ограничение: обращение к функции не должно переопределять значение переменной, фигурирующей в том же операторе, или влиять на результат другой функции, вызываемой в том же операторе. Например, в

```
d = max(dist(p, q), dist(q, r))
```

функция *dist* не должна переопределять переменную *q*.

Подобных эффектов можно избежать, если программировать процедуру в виде функции лишь в том случае, когда в процедуре только один выходной параметр.

8.12. Перегрузка и родовые интерфейсы

8.12.1. Перегрузка процедур

Иногда полезно иметь возможность обращаться к нескольким процедурам при помощи одного имени. Реализовать подобную возможность можно, объединяя посредством *родового интерфейса* различные процедуры под одним *родовым именем*. Имена объединяемых процедур называются *специфическими*. Сам же механизм вызова разных процедур под одним именем называется *перегрузкой*. Механизм перегрузки реализован при разработке встроенных процедур (см. разд. 6.3).

Построим, например, функцию *mymax*(*arg*1, *arg*2), возвращающую максимальное значение из двух ее параметров. Параметры функции должны быть одного из следующих типов: INTEGER(4), REAL(4) или CHARACTER(*). Тип результата функции совпадает с типом параметров.

На самом деле для каждого типа параметров придется создать свою функцию, например *inmax*, *remax* и *chmax*, а затем, применив родовой интерфейс, мы объединим созданные функции под одним родовым именем.

```
program getest
 interface mymax                  ! Задание родового интерфейса
  function inmax (int1, int2)     ! mymax - родовое имя для функций
   integer(4) inmax, int1, int2   ! inmax, remax, chmax
  end function inmax
  function remax (re1, re2)       ! inmax, remax, chmax - специфические
   real(4) remax, re1, re2        ! имена функций, объединенных под
  end function remax              ! одним родовым именем mymax
  function chmax (ch1, ch2)
   character(*) ch1, ch2
   character(len = max(len(ch1), len(ch2))) chmax
  end function chmax
 end interface
 integer(4) :: ia = 1, ib = 2
 real(4) :: ra = -1, rb = -2
 character(5) :: cha = 'abcde', chb = 'ABCDE'
 print *, mymax(ia, ib)           !            2
 print *, mymax(ra, rb)           !      -1.000000
 print *, mymax(cha, chb)         !         abcde
end program getest
function inmax(int1, int2)
 integer(4) inmax, int1, int2
 if(int1 >= int2) then
  inmax = int1
 else
  inmax = int2
 end if
end function inmax
```

```
function remax(re1, re2)
 real(4) remax, re1, re2
 if(re1 >= re2) then
  remax = re1
 else
  remax = re2
 end if
end function remax

function chmax(cha1, cha2)
 character(*) cha1, cha2
 character(len = max(len(cha1), len(cha2))) chmax
 if(lge(cha1, cha2)) then
  chmax = cha1
 else
  chmax = cha2
 end if
end function chmax
```

Родовой интерфейс можно задать более компактно, если функции являются модульными процедурами. В этом случае интерфейс к ним задан явно и в создаваемый для родового имени интерфейсный блок вставляется оператор

MODULE PROCEDURE *список имен процедур*

в котором перечисляются имена всех модульных процедур для перегрузки. Например:

```
module gemod
 interface mymax
  module procedure inmax, remax, chmax
 end interface
 contains
  function inmax(int1, int2)
   ...
  end function inmax
  function remax(re1, re2)
   ...
  end function remax
  function chmax(cha1, cha2)
   ...
  end function chmax
end module gemod

program getest
 use gemod
 print *, chmax('abcde', 'ABCDE')    ! Вызов процедуры можно выполнить,
end program getest                   ! используя ее специфическое имя
```

Замечание. Фортран 95 позволяет завершать интерфейсный блок родовым именем. Например:

```
interface mymax                  ! Задание родового интерфейса
  module procedure inmax, remax, chmax
end interface mymax              ! Завершаем блок родовым именем mymax
```

Ту же форму задания интерфейса можно применить и в том случае, если в модуле содержатся только объединяемые под родовым именем процедуры:

```
program getest
 use fuma                        ! Модуль fuma содержит функции inmax, remax,
 interface mymax                 ! chmax, но не содержит родового интерфейса
  module procedure inmax, remax, chmax
 end interface
 print *, chmax('abcde', 'ABCDE')    ! abcde
end program getest
```

Объединяемые процедуры могут иметь разное число параметров. Также под одним именем могут быть перегружены и подпрограммы и функции.

В интерфейсном блоке родовое имя может совпадать с любым специфическим именем процедуры этого блока.

Родовое имя может также совпадать с другим доступным, например через *use*-ассоциирование, родовым именем. Тогда с помощью этого имени могут быть вызваны все охватываемые им процедуры.

Все процедуры, объединяемые под одним родовым именем, должны различаться настолько, чтобы для каждого конкретного вызова можно было однозначно выбрать одну из объединенных процедур. Для этого в каждой паре перегружаемых процедур хотя бы одна должна иметь обязательный параметр, удовлетворяющий сразу двум условиям:

- по своему положению в списке параметров он либо вообще не имеет аналога среди формальных параметров другой процедуры, либо соответствует параметру, имеющему другой тип или разновидность типа или другой ранг;
- формальный параметр с таким же именем либо отсутствует в другой процедуре, либо присутствует, но имеет другой тип или разновидность типа или другой ранг.

Пример нарушения второго условия:

```
interface fu12
 function f1(x, i)               ! Каждый из формальных параметров f1 имеет
  real f1, x                     ! аналог среди формальных параметров функции f2
  integer i                      ! И наоборот, каждый из формальных параметров f2
 end function f1                 ! имеет аналог среди формальных параметров f1
 function f2(i, x)
  real f2, x
  integer i
```

```
 end function f2
end interface
```

Компилятор CVF, получивший такие интерфейсы, выдаст сообщение: Error: The type/rank/keyword signature for this specific procedure matches another specific procedure that shares the same generic-name. [F2] function f2(i, x).

8.12.2. Перегрузка операций и присваивания

Область применения встроенной операции можно расширить. Это выполняется при помощи интерфейсного блока, заголовок которого имеет вид:

INTERFACE OPERATOR(*задаваемая операция*)

Все остальные компоненты интерфейсного блока такие же, как и при перегрузке процедур. Задающая операцию процедура должна обязательно быть функцией с одним (в случае унарной операции) или двумя параметрами, имеющими вид связи IN. Параметры функции должны быть обязательными. Результатом функции не может быть перенимающая длину строка.

Такой блок связывает задаваемую операцию с одной или несколькими функциями, задающими выполняемые этой операцией действия. Например, можно задать операцию сложения переменных производного типа. Тогда в случае применения операции в зависимости от типа ее операндов будет выполняться либо встроенная операция сложения, если операнды числовые, либо заданная операция, если типы операндов совпадают с типами формальных параметров заданной в интерфейсном блоке функции. Изложенный механизм задания операции называется *перегрузкой операции*.

```
module tdes                    ! Демонстрация перегрузки операции сложения
 type ire
  integer a                    ! Перегрузка операций для производных типов
  real ra                      ! необходима, поскольку для них не существует
 end type ire                  ! ни одной встроенной операции
end module tdes

module plup
 use tdes
 interface operator(+)         ! Задание операции сложения для типа ire
  module procedure funir
 end interface
 contains
 function funir(rec1, rec2)    ! Параметры задающей операцию функции
  type(ire) funir              ! должны иметь вид связи IN
  type(ire), intent(in) :: rec1, rec2
  funir = ire(rec1.a + rec2.a, rec1.ra + rec2.ra)
 end function funir
end module plup
```

```
program top
use plup                          ! Тип ire передается через модули plup - tdes
integer :: ia = 1, ib = -1, ic
type(ire) :: t1 = ire(1, 1.0), t2 = ire(2, 2.0), t3 = ire(3, 3.0), t4
ic = ia + ib                      ! Выполняется встроенная операция сложения
t4 = t1 + t2 + t3                 ! Выполняется заданная операция сложения
print *, ic, t4                   !        0        6        6.000000
end
```

При перегрузке встроенной операции нельзя изменять число ее операндов. Так, нельзя задать унарную операцию умножения. Поскольку операции отношения имеют две формы, например (.LE. и <=), то заданный для них интерфейс распространяется на каждую из форм.

Таким же образом можно ввести и новую операцию. Имя вводимой операции должно обрамляться точками. Например, для обозначения операции сложения переменных типа *ire* можно было бы ввести операцию .plus.:

```
interface operator(.plus.)
```

Тогда применение вновь введенной операции может быть таким:

```
t3 = t1 .plus. t2 .plus. t3
```

Как и встроенная, вновь вводимая операция может быть распространена на разные типы операндов.

Также можно выполнить *перегрузку присваивания*. Интерфейсный блок при перегрузке присваивания имеет заголовок

INTERFACE ASSIGNMENT(=)

Все остальные компоненты интерфейсного блока такие же, как и при перегрузке процедур. Задающая присваивание процедура должна обязательно быть подпрограммой с двумя формальными параметрами, первый из которых имеет вид связи OUT или INOUT, а второй - IN. Параметры подпрограммы должны быть обязательными. Первый параметр подпрограммы в результате выполнения заданного присваивания будет содержать его результат, во второй - передается значение правой части присваивания.

Пример. Задать присваивание для выполнения инициализации производного типа данных.

```
module tic
 type icha
  integer a, b
  character(10) fi, se
 end type icha
end module tic

module oves
 use tic
```

```
 interface assignment(=)          ! Задание присваивания для инициализации записи
  module procedure assir
 end interface
 contains
 subroutine assir(rec, k)
  type(icha), intent(out) :: rec
  integer, intent(in) :: k
  integer :: stlen
  stlen = len(rec%fi)
  rec = icha(k, k, repeat(char(k), stlen), repeat(char(k), stlen))
 end subroutine assir
end module oves

program top
 use oves                         ! Тип ire доступен посредством use-ассоциирования
 type(icha) :: t                  ! через модули oves - tic
 t = 35                           ! Выполняется заданное присваивание
 print '(2i4, 2(1x,a))', t        ! 35  35 ########## ##########
end
```

Если две процедуры, задающие одну родовую операцию или присваивание, имеют одно и то же число обязательных параметров, то для однозначности вызова одна из них должна иметь по крайней мере один формальный параметр, который по своему положению в списке параметров соответствует параметру другой процедуры, имеющему либо другой тип, либо другую разновидность типа, либо другой ранг. Это правило распространяется на случай, когда более двух процедур имеют одну родовую операцию или задают присваивание.

8.12.3. Общий вид оператора INTERFACE

Оператор INTERFACE применяется для явного задания интерфейса к внешней процедуре, родового интерфейса, родовой операции и присваивания. Его синтаксис:

INTERFACE [*родовое описание*]
  [*тело интерфейса*]
  ...
  [MODULE PROCEDURE *список имен процедур*]
  ...
END INTERFACE

где *родовое описание* - это *родовое имя*, или

OPERATOR(*определяемая операция*)

или

ASSIGNMENT(=)

тело интерфейса - задает характеристики *внешних* или *формальных* процедур и представляет в случае функции

заголовок функции
[*раздел описаний*]
END [FUNCTION [*имя функции*]]

либо в случае подпрограммы

заголовок подпрограммы
[*раздел описаний*]
END [SUBROUTINE [*имя подпрограммы*]]

Оператор MODULE PROCEDURE может появляться в интерфейсном блоке, лишь когда присутствует *родовое описание*. При этом все процедуры *списка имен процедур* должны быть доступными модульными процедурами. Характеристики модульных процедур не должны появляться в интерфейсном блоке.

Процедура, объявленная в интерфейсном блоке, обладает атрибутом EXTERNAL. Она не может быть объявлена внешней посредством атрибута или оператора EXTERNAL в программной единице, в которой присутствует или доступен через *use*-ассоциирование интерфейсный блок с этой процедурой. Для процедуры в блоке видимости может быть задан лишь один интерфейсный блок.

Внутренние, модульные и встроенные процедуры имеют явно заданный интерфейс, и их имена не должны появляться в интерфейсном блоке. Исключение составляет случай, когда необходимо задать для них родовое имя. При этом имена модульных процедур задаются оператором MODULE PROCEDURE. Если же имя объявленной в интерфейсном блоке процедуры совпадает с именем встроенной процедуры, то такая встроенная процедура становится недоступной. В то же время должна существовать внешняя процедура с таким же именем.

Если имя процедуры интерфейсного блока совпадает с именем формального параметра процедуры, в которой задан интерфейсный блок, то такой формальный параметр является формальной процедурой.

Тело интерфейса не может содержать операторы ENTRY, DATA, FORMAT, объявления операторных функций. Можно, правда, задать самостоятельный ENTRY-интерфейс, использовав в теле интерфейса имя точки входа в качестве имени процедуры.

Программная единица BLOCK DATA не может содержать интерфейсный блок.

8.13. Ассоциирование имен

Большинство объектов Фортран-программы являются локальными. К ним относятся имена переменных, констант, производных типов данных, внутренних и модульных процедур, операторных функций. Глобальными

являются имена главной программы, встроенных и внешних процедур, модулей, *common*-блоков, BLOCK DATA. Однако локальный объект программной единицы можно сделать доступным в другой программной единице, используя *ассоциирование имен*: ассоциирование параметров процедуры, *use*-ассоциирование и ассоциирование через носитель.

В момент вызова процедуры между формальными и фактическими параметрами устанавливается связь, или, иными словами, формальные параметры процедуры *ассоциируются* с фактическими. Благодаря такой связи:

- осуществляется обмен данными между программными единицами;
- реализуется альтернативный возврат из подпрограммы (разд. 8.19);
- передается в процедуру имя внешней или встроенной функции.

Более подробно о правилах соответствия формальных и фактических параметров см. в разд. 8.11.1.

Объекты модуля становятся доступны в программной единице, если в ней есть оператор USE, содержащий имя модуля. Использование этого оператора равнозначно повторному описанию всех не имеющих атрибута PRIVATE объектов модуля внутри программной единицы с сохранением всех имен (если нет переименований) и свойств. (Доступ к объектам модуля может быть ограничен за счет использования в операторе USE опции ONLY.) В этом случае говорится, что объекты модуля доступны за счет *use*-ассоциирования. Благодаря *use*-ассоциированию может быть обеспечен доступ к следующим объектам модуля:

- именованным объектам данных;
- определениям производных типов;
- интерфейсным блокам;
- модульным процедурам;
- родовым интерфейсам;
- *namelist*-группам.

Use-ассоциирование передает как данные, так и статус объектов, например статус размещаемого массива (разд. 4.8.2).

При подключении к программной единице модулей нельзя допускать дублирования передаваемых через *use*-ассоциирование имен и локальных имен самой программной единицы. Однако доступные через *use*-ассоциирование имена могут совпадать, если:

- нет ни одного использования дублированного имени;
- продублированное имя является родовым (разд. 8.12).

Избежать конфликтов имен можно за счет переименования, использования опции ONLY и за счет придания атрибута PRIVATE тем объектам модуля, которые предназначены только для внутреннего использования в модуле.

В модульных и внутренних процедурах доступны все объекты носителя этих процедур, в том числе и объекты, доступные носителю через *use*-ассоциирование. Такой механизм доступа к объектам носителя называется *ассоциированием через носитель*. Общее правило таково: имя объекта носителя считается повторно описанным с теми же свойствами в модульной или внутренней процедуре при условии, что в процедуре нет другого объекта с таким же именем, объявленного локально, или доступного путем *use*-ассоциирования, или являющегося локальным формальным параметром или результирующей переменной.

Пример:

```
real :: x = 1.0, w = 1.0, z = 1.0
 call decar(x)
 print '(3f5.2)', x, w, z            ! 5.0  1.0  5.0
 contains
  subroutine decar(x)
   real x, w                        ! Локальная переменная w подпрограммы decar
   x = 5; w = 5; z = 5              ! закрывает локальную переменную w носителя
  end subroutine
end
```

8.14. Область видимости имен

Областью видимости именованного объекта называется часть программы, в которой можно ссылаться на этот объект.

Например, на объявленный во внешней процедуре объект можно ссылаться в этой процедуре. Кроме того, этот объект за счет ассоциирования через носитель доступен в любой ее внутренней процедуре. При этом в самой процедуре могут существовать фрагменты, в которых ссылка на эту переменную невозможна. Такими фрагментами могут быть определения типов и интерфейсные блоки. Приведем фрагмент процедуры для описанной ситуации.

```
subroutine reg( )
 integer, parameter :: m = 40, n = 20      ! Область 1
 real a(m, n), b(3, 4, 5)                  ! Область 1
 type win
  integer n                                ! Область 2
  real a(n)                                ! Область 2
 end type
 interface
  subroutine ones(a, m)                    ! Область 3
   integer m                               ! Область 3
   real a(:, :, :)                         ! Область 3
  end subroutine ones                      ! Область 3
 end interface
 type(win) tin(n)                          ! Область 1
```

```
 a = real(n)                          ! Область 1
 tin(n).a = real(m)                   ! Область 1
 print *, a(m, n), tin(n).a(1)        ! Область 1
 call ones(b, n)                      ! Область 1
 call two( )
 contains
  subroutine two( )          ! Внутренняя подпрограмма не является областью
   integer m                 ! видимости объектов носителя, но объекты носителя
   m = n                     ! n, a, b, win, tin и интерфейс к подпрограмме
   a = real(m)               ! ones достижимы в two благодаря ассоциированию
   call ones(b, m)           ! через носитель. Переменная m подпрограммы
  end subroutine two         ! two закрывает константу m носителя
end subroutine reg

program t2                   ! Драйвер подпрограммы reg
 call reg( )
end program t2

subroutine ones(a, m)
 integer m
 real a(:, :, :)
 a = real(m)
end subroutine ones
```

В этом фрагменте область 1 является областью видимости констант *m* и *n*. Впрочем, эти же константы видны и в области 2 определения производного типа: в объявлении REAL *a*(*n*) используется константа *n* из области 1. В то же время в области 2 можно объявить компонент с именем *n*. В области 3 интерфейсного блока константы *m* и *n*, так же как и массив *a*(1:*m*, 1:*n*), не видны. Иными словами, использованные в областях 1, 2 и 3 имена *m*, *n* и *a* относятся к разным объектам данных. Таким образом, область видимости констант *m* и *n* состоит из трех блоков видимости, разделенных областью 3. Областью видимости массива *a*(1:*m*, 1:*n*) и трехмерного массива *b* являются два блока, разделенные уже двумя областями с номерами 2 и 3.

Область видимости именованного объекта зависит от вида его имени. Имена объектов разделяются на *глобальные*, *локальные* и *операторные*.

Глобальными являются имена главной программы, модулей, встроенных и внешних процедур и *common*-блоков. Эти имена известны в любой программной единице, и не может быть двух глобальных объектов с одним именем. Так, не может быть *common*-блока с именем *sqrt*, поскольку это имя принадлежит встроенной функции.

Если же в каком-либо блоке видимости определена локальная переменная *sqrt*, то глобальное имя встроенной функции SQRT в этом блоке видимости становится недоступным.

Если же глобальное имя встроенной процедуры определено в блоке видимости с атрибутом EXTERNAL, то встроенная процедура также становится недоступной в этом блоке видимости, но введенное имя трактуется как глобальное имя внешней процедуры.

Пример:

```
real :: sqrt, x = 4.0, y
real, external :: sin          ! Должна быть определена внешняя функция sin
sqrt = 5.0                     ! Локальное имя закрывает глобальное имя
y = sqrt(x)                    ! Ошибка - встроенная функция SQRT недоступна
```

К *локальным* именам относятся имена переменных, формальных параметров, именованных констант, производных типов, операторных функций, внутренних, модульных и формальных процедур, родовых описаний, *namelist*-групп. Блоками видимости локальных имен являются:

- определение производного типа;
- тело интерфейса, за исключением содержащихся в нем определений производных типов и тел интерфейсных блоков;
- программная единица, за исключением содержащихся в ней определений производных типов, интерфейсных блоков и внутренних процедур.

Локальные имена в случае объявления их в блоке видимости закрывают имена глобальных объектов, и последние становятся недоступными в этом блоке видимости. Исключение составляют применяемые при вызове процедур ключевые слова, родовые описания и имена *common*-блоков. Выше было показано, как локальное имя *sqrt* закрыло глобальное имя встроенной функции SQRT. Такой же эффект вызовет и использование внутренней функции, например, с именем *tan*, которое закроет в блоке видимости имя встроенной функции TAN, например:

```
subroutine dehi
 ...
 y = tan(x)                    ! Будет вызвана внутренняя функция tan
 contains
  function tan(x)
  ...
  end function tan
end subroutine dehi
```

Областью видимости операторного имени является один оператор. Операторные имена могут появляться при задании операторной функции, а также в неявных циклах операторов DATA и конструкторов массивов. Областью видимости формальных параметров операторной функции является оператор задания этой функции. Областью видимости переменной неявного цикла, которая должна быть целого типа, является этот цикл.

Параметры неявных циклов операторов В/В не являются операторными, а относятся к локальным и могут быть вещественного типа. Например:

```
real c(100)
z(x, y) = sin(x) * exp(-y)              ! Определяем операторную функцию z
b = 55.0
! Переменная b будет использована в качестве параметра цикла оператора WRITE
write(*, '(5f7.4)') ((z(a, b), a = 0.0, 1.0, 0.2), b = 0.0, 1.0, 0.2)
k = 55                                  ! k - локальная переменная
c = (/ (float(k), k = 1, 100) /)        ! k - пример операторного имени
print *, b                              !       1.20000
print *, k                              !       55
end
```

Замечание. Использовать нецелые параметры в неявном цикле, так же как и в DO-цикле с параметром, не рекомендуется (см. П.-2.2.4).

Имя локального объекта не закрывает имени *common*-блока, поэтому эти имена могут быть одновременно использованы в блоке видимости. Имя *common*-блока, если оно используется в операторе SAVE, должно обрамляться слешами.

Например:

```
common /vab/ a, b
real vab                    ! Имена переменной vab и common-блока совпадают
save :: /vab/, vab          ! Атрибут SAVE имеют и переменная и common-блок
```

Имена локальных объектов в блоке видимости могут совпадать с используемыми при вызовах процедур ключевыми словами. Область видимости ключевых слов определяется областью видимости интерфейсного блока к процедуре, в которой эти ключевые слова описаны. Область действия интерфейсного блока может быть распространена на другую программную единицу в результате *use*-ассоциирования или ассоциирования через носитель.

8.15. Область видимости меток

Метки являются локальными объектами. Главная программа и каждая процедура имеют свой независимый набор меток. Оператор END носителя может иметь метку. Если в таком носителе есть внутренние процедуры, то они разбивают область видимости этой метки на два блока: до оператора CONTAINS и оператор END носителя.

8.16. Ассоциирование памяти

Фортран предоставляет пользователю еще один способ обмена данными - это ассоциирование памяти. Применив оператор COMMON, в программе можно создать общую область памяти, на которую можно ссылаться изо всех программных единиц, содержащих этот оператор. Например:

```
program gocom
 integer(4) a, b, c
 common /vab/ a, b, c          ! Создаем общую область числовой памяти
 a = -1                        ! В этой области существует 3 единицы
 call chaco( )                 ! памяти по 4 байта каждая
 print *, a, b, c              !       1       2       3
end program gocom

subroutine chaco( )
 integer(4) ia, ib, ic
 common /vab/ ia, ib, ic
 print *, ia                   !       -1
 ia = 1;  ib = 2;   ic = 3
end subroutine
```

В приведенном примере переменные *a*, *b* и *c* главной программы и переменные *ia*, *ib* и *ic* подпрограммы *chaco* используют одну и ту же область памяти. Более того, переменные *a* и *ia* адресуют одну и ту же единицу памяти. Это означает, что после выхода из подпрограммы переменная *a* получит значение переменной *ia*. То же справедливо и для пар переменных *b* и *ib*, *c* и *ic*.

Одинаковые по порядку имена в расположенном в разных программных единицах *common*-блоке могут совпадать. Они также могут и различаться (это видно из примера). Более того, они могут различаться рангом и формой. Так, мы получим тот же результат, создав, например, подпрограмму:

```
subroutine chaco( )
 integer(4) abc, k
 common /vab/ abc(3)           ! Массив abc вместо переменных a, b и c
 print *, abc(1)               !       -1
 abc = (/ (k, k = 1, 3) /)
end subroutine
```

В этом случае уже *a* и *abc*(1) адресуют одну и ту же единицу памяти. То же справедливо и для пар *b* и *abc*(2), *c* и *abc*(3).

Иной пример ассоциирования памяти - применение оператора EQUIVALENCE, который явно указывает, что два или более объекта занимают одну область памяти.

```
integer :: a(5) = 3, b(5)
equivalence(a, b)
print *, a                     !       3       3       3       3       3
print *, b                     !       3       3       3       3       3
```

Рассмотренный на примерах механизм доступа к памяти называется *ассоциированием памяти*. Такой механизм используется для обмена данными. Правда, в прежние времена при недостатке вычислительных ресурсов он часто использовался и для экономии памяти. Последнее выполнялось за счет применения оператора EQUIVALENCE. Однако такая

практика является причиной многих ошибок и не может быть рекомендована для применения (прил. 2).

Для дальнейшего рассмотрения вопроса нам понадобятся некоторые дополнительные сведения.

8.16.1. Типы ассоциируемой памяти

Под *единицей памяти* понимают область памяти компьютера, выделяемую под определенные данные. Размер такой единицы зависит от типа и параметра разновидности типа. Так, единица памяти под скаляр типа REAL(4) равна 4 байтам, а скаляр типа COMPLEX(8) занимает две единицы памяти по 8 байт каждая.

Единица памяти может быть:

- числовой;
- текстовой;
- неспецифицированной.

Числовая единица памяти выделяется под нессылочный скаляр (т. е. скаляр без атрибута POINTER) стандартного вещественного, целого или логического типа.

Текстовую единицу памяти занимает нессылочный скаляр стандартного символьного типа единичной длины.

К объектам производного типа ассоциирование памяти применимо лишь при наличии у них атрибута SEQUENCE. Если в определении типа использованы другие производные типы, то они тоже должны иметь атрибут SEQUENCE. В таком случае объекты производного типа могут быть использованы в операторах COMMON, EQUIVALENCE и в качестве параметров процедур.

С производным типом, имеющим атрибут SEQUENCE и не имеющим ссылочных компонентов на любом уровне, ассоциируется:

- *числовая память*, если конечные компоненты типа относятся к стандартному целому, вещественному, вещественному двойной точности, комплексному или логическому типу;
- *текстовая память*, если конечные компоненты типа относятся к стандартному символьному типу.

Неспецифицированная единица памяти присуща любым другим производным типам с атрибутом SEQUENCE, а также объектам с атрибутом POINTER. Размер неспецифицированной единицы памяти таких объектов уникален для каждого типа, параметра типа и ранга.

Нессылочный массив встроенного типа или производного типа с атрибутом SEQUENCE занимает ряд последовательных *отрезков памяти*, по одному на каждый элемент массива в порядке их следования в массиве. Нессылочный скаляр производного типа с атрибутом SEQUENCE, имеющий *n* конечных

компонентов, занимает *n отрезков памяти*, по одному на каждый конечный компонент в порядке их объявления в производном типе.

Последовательность отрезков и единиц памяти образует *объединенный отрезок памяти*.

Для правильного обмена данными следует ассоциировать объекты с единицами памяти одного и того же типа.

8.16.2. Оператор COMMON

Оператор COMMON создает общую область памяти - глобальный отрезок памяти, доступный в различных программных единицах.

COMMON [/[*cname*]/] *список имен* [[,] /[*cname*]/ *список имен*] ...

cname - имя общего блока (*common*-блока), которому принадлежат объекты соответствующего *списка имен*. Имя может быть опущено. Такой *common*-блок называется *неименованным*. Если первый задаваемый в операторе COMMON общий блок является неименованным, то слеши могут быть опущены, например:

```
common a, r, g(40)
```

Имя *common*-блока является глобальным и должно отличаться от любого другого глобального имени (программной единицы, другого *common*-блока), но может совпадать с именем локального объекта, кроме именованной константы.

список имен - список входящих в именованную или неименованную общую область имен простых переменных, строк, записей, массивов и объявлений массивов. При объявлении в *common*-блоке массива размеры его границ задаются в виде целочисленных констант или константных выражений. Объекты *common*-блока могут иметь атрибуты POINTER и TARGET. Имена в списке разделяются запятыми. Каждое имя в программной единице может появляться в *списке имен* только один раз и не может появляться в другом *списке имен* этой программной единицы. В *списке имен* не могут появляться имена формальных параметров, процедур, точек входа, результирующей переменной функции, размещаемых массивов и автоматических объектов, именованных констант (объектов с атрибутом PARAMETER). Объекты производного типа могут быть помещены в *common*-блок при наличии у них атрибута SEQUENCE.

Оператор COMMON размещается в разделе объявлений программной единицы. В программной единице можно объявить несколько общих областей, задаваемых одним или несколькими операторами COMMON.

Имя любого *common*-блока (включая и пустое имя) может появляться в разделе описаний программного модуля более одного раза. При этом список элементов конкретного *common*-блока рассматривается как продолжение списка элементов предшествующего *common*-блока с тем же именем.

Пример:

```
common x, y, /com1/ a, b, // z(15)
common /com1/ c(22)
```

В программе будут заданы два *common*-блока: неименованный, в который войдут переменные *x*, *y* и массив *z*, и именованный - *com*1, содержащий переменные *a*, *b* и массив *c*. Конечно, для данного случая следовало бы задать *common*-блоки более наглядно:

```
common x, y, z(15)                ! Неименованный common-блок
common /com1/ a, b, c(22)
```

В разных программных единицах переменные одного *common*-блока ассоциируются с одним и тем же отрезком памяти. Порядок размещения в оперативной памяти элементов *common*-блока совпадает с порядком их следования в операторе COMMON.

Длина общей области равна числу байт памяти, необходимых для размещения всех ее элементов, включая расширения за счет EQUIVALENCE-ассоциирования (прил. 2). Если несколько разных программных единиц обращаются к одному именованному *common*-блоку, то в каждой из них *common*-блок должен иметь одну и ту же длину. Неименованный *common*-блок в разных программных единицах может иметь разную длину. Длина неименованного *common*-блока равна длине наибольшего существующего в программе неименованного *common*-блока.

Фортран максимально уплотняет размещение переменных в памяти компьютера. При этом переменные *common*-блока размещаются в памяти по следующим правилам:

- переменные типа BYTE, INTEGER(1), LOGICAL(1) или CHARACTER размещаются без промежутков сразу после предшествующей переменной *списка имен*. То же справедливо для переменных производного типа размеров в 1 байт;
- все другие простые переменные и несимвольные массивы начинаются на следующем четном байте, ближайшем к предыдущей переменной;
- символьные массивы всегда начинаются на следующем свободном байте;
- элементы любого массива следуют один за другим без промежутков;
- все *common*-блоки начинаются на байте, номер которого кратен четырем.

Замечание. Программисты, использующие ассоциирование памяти, могут повысить быстродействие программ, правильно размещая переменные в *common*-блоке. Детально этот вопрос обсуждается в [1].

Из-за разных принципов выравнивания символьных и несимвольных переменных в памяти ЭВМ одновременное применение символьных переменных нечетной длины и несимвольных переменных в одном *common*-блоке может привести к проблемам. Так, если такому смешанному *common*-

блоку в другой программной единице соответствует *common*-блок, содержащий только несимвольные переменные, то возникнут не используемые при ассоциировании байты общей области памяти. Чтобы избежать подобных явлений, не следует смешивать в одном *common*-блоке символьные и несимвольные данные.

С переменными *common*-блока можно использовать только два атрибута: ALIAS и C. Не следует из-за приведенных проблем выравнивания использовать *common*-блок для доступа к структурам СИ, применяя вместо него определение типа с атрибутом EXTERN.

Инициализация элементов именованных *common*-блоков выполняется в программной единице BLOCK DATA. Переменные, включенные в *список имен common*-блока, не могут быть инициализированы в операторе DATA за исключением того случая, когда оператор DATA использован в программной единице BLOCK DATA. Больше того, переменная *common*-блока не может быть инициализирована и в операторе объявления типа.

Атрибут SAVE не может быть дан отдельной переменной *common*-блока, но может быть задан блоку целиком. Имя *common*-блока при этом обрамляется слешами, например:

```
save /com1/                    ! com1 - имя common-блока
```

Неименованный *common*-блок отличается от именованного следующими свойствами:

- после выполнения в процедуре операторов RETURN или END объекты именованного *common*-блока становятся неопределенными, если только *common*-блок не имеет атрибута SAVE. Объекты неименованного *common*-блока всегда сохраняют свои значения после выполнения RETURN или END;
- именованный *common*-блок должен иметь одну и ту же длину во всех его использующих программных единицах. Длина неименованного *common*-блока может быть разной в разных программных единицах;
- объекты неименованного *common*-блока нельзя инициализировать в программной единице BLOCK DATA.

Замечание. В Фортране все объекты (кроме автоматических) по умолчанию имеют атрибут SAVE. Поэтому явное задание этого атрибута именованному *common*-блоку полезно при создании переносимых на другие платформы программ.

Common-блок может быть объявлен в модуле. Тогда его описание не должно появляться в использующей модуль программной единице.

При работе с *common*-блоками:

- следует делать все описания данного блока одинаковыми во всех использующих его программных единицах;

- следует избегать смешения в одном *common*-блоке разнотипных единиц памяти (из-за описанных выше проблем выравнивания).

```
program gocom                  ! Допустимое, но нерекомендуемое различие
 complex (4) z                 ! описаний common-блока в разных
 common /vab/ z                ! программных единицах
 call chaco ( )
 print *, z                    !        (5.000000, -5.000000)
end program gocom

subroutine chaco ( )
 real(4) x, y
 common /vab/ x, y             ! Вещественные переменные x и y вместо
 x = 5.0; y = -5.0             ! комплексной переменной z
end subroutine
```

Замечание. В современном Фортране *common*-блоки могут быть полностью заменены модулями.

8.16.3. Программная единица BLOCK DATA

При необходимости начальные значения элементов *именованного common-блока* можно задать, используя программную единицу BLOCK DATA. Ее общий вид:

BLOCK DATA [*имя блока данных*]
раздел объявлений
операторы DATA *задания начальных значений элементов общей области* &
END [BLOCK DATA [*имя блока данных*]]

имя блока данных является глобальным именем и не может совпадать с локальным именем переменной блока данных и с другим глобальным именем.

В программе может быть определено несколько программных единиц BLOCK DATA, имеющих различные имена и выполняющих инициализацию элементов разных именованных *common*-блоков. Причем в программе может быть задана только одна неименованная программная единица BLOCK DATA.

В одной программной единице BLOCK DATA может появляться несколько разных именованных *common*-блоков. Один и тот же *common*-блок не может появляться в разных программных единицах BLOCK DATA. Не могут быть инициализированы в BLOCK DATA объекты с атрибутом POINTER.

В BLOCK DATA могут быть использованы только следующие операторы: USE, IMPLICIT, COMMON, DATA, END, DIMENSION, EQUIVALENCE, POINTER, TARGET, MAP, PARAMETER, RECORD,

SAVE, STRUCTURE, UNION - и операторы *объявления типа*. Использование исполняемых операторов в BLOCK DATA недопустимо.

Присутствующие в BLOCK DATA операторы объявления типа не могут содержать атрибуты ALLOCATABLE, EXTERNAL, INTENT, OPTIONAL, PRIVATE и PUBLIC.

Имя блока данных может появляться в операторе EXTERNAL. Это позволит при сборке программы загрузить из библиотеки нужный BLOCK DATA.

Пример:

```
block data bd2                  ! Этот блок может следовать сразу за
 complex z                      ! программой gocom предыдущего примера
 common /vab/ z
 data z /(2.0, 2.0)/            ! Инициализация объекта common-блока
end block data bd2
```

8.17. Рекурсивные процедуры

Фортран поддерживает рекурсивные вызовы внешних, модульных и внутренних процедур. Процедура называется *рекурсивной*, если она обращается сама к себе или вызывает другую процедуру, которая, в свою очередь, вызывает первую процедуру. В первом случае рекурсия называется *прямой*, во втором - *косвенной*.

Процедура также является рекурсивной, если содержит оператор ENTRY и обращается к любой задаваемой этим оператором процедуре.

Оператор объявления рекурсивной процедуры должен предваряться префиксом RECURSIVE. Внутри рекурсивной процедуры интерфейс к этой процедуре является явным.

Пример. Разработать подпрограмму *subst*, которая в данной строке заменяет все вхождения подстроки *sub*1 на подстроку *sub*2. Так, если дана строка 'abc1abc2abc3' и *sub*1 = 'abc', а *sub*2 = ' d', то результатом должна быть строка ' d1 d2 d3'.

```
program stgo
 character(len = 20) :: st = 'abc1abc2abc3'
 call subst(st, 'abc', ' d')          ! subst содержит прямую рекурсию
 write(*, *)  st                      ! d1 d2 d3
end

recursive subroutine subst(st, sub1, sub2)
 character(len = *) st, sub1, sub2 ! Длина каждой строки определяется
 integer ip                        ! длиной соответствующего
 ip = index(st, sub1)              ! фактического параметра
 if(ip > 0) then
  st = st(:ip - 1) // sub2 // st(ip + len(sub1):)
  call subst(st, sub1, sub2)       ! Рекурсивный вызов подпрограммы
```

```
 end if                          ! выполняется до тех пор, пока не
end                              ! выполнены все замены sub1 на sub2
```

Если функция содержит прямую рекурсии, т. е. непосредственно вызывает сама себя, результату необходимо дать имя, отличное от имени функции. Это выполняется путем добавления в заголовок функции предложения RESULT. В случае косвенной рекурсии имя результирующей переменной и имя функции могут совпадать.

Пример. Вычислить факториал числа *n*.

```
program fact
 integer n /5/, ifact
 write(*, *) ' 5! = ', ifact(n)          ! 5! = 120
end

recursive function ifact(n) result (fav)
 integer fav                             ! В операторе объявления используется
 integer, intent(in) :: n                ! не имя функции ifact, а имя результата fav
 if(n <= 1) then
  fav = 1
 else
  fav = n * ifact(n - 1)                 ! Рекурсия продолжается, пока n > 1
 end if
end
```

Тип результата рекурсивной функции можно задать и в ее заголовке, например:

```
recursive integer function ifact(n) result (fav)
```

или:

```
integer recursive function ifact(n) result (fav)
```

Рекурсивная процедура обязательно должна содержать проверку, ограничивающую число рекурсивных вызовов.

8.18. Формальные процедуры

Имя внешней, модульной процедуры и встроенной функции можно использовать в качестве фактического параметра процедуры. В этом случае соответствующий формальный параметр называется *формальной процедурой*.

Формальные процедуры используются в задачах, решаемых для разных функций. Например, поиск экстремума, корня уравнения, вычисление определенного интеграла и т. д. В таких случаях создается процедура решения типовой задачи для широкого класса функций, в которую конкретная функция передается как фактический параметр.

Имя рассматривается как имя внешней процедуры, если оно обладает атрибутом EXTERNAL. И рассматривается как имя встроенной процедуры, если имеет атрибут INTRINSIC. Если это имя используется в качестве фактического параметра процедуры, то соответствующим формальным

параметром должно быть имя формальной процедуры. Формальная процедура, если она является функцией, должна иметь тот же тип и разновидность типа, что и фактическая функция. Формальная и фактическая процедуры должны быть согласованы по числу, типу и рангу используемых в них параметров.

Атрибуты EXTERNAL и INTRINSIC могут иметь и иное применение. В частности, можно описать с атрибутом INTRINSIC все используемые в блоке видимости встроенные процедуры, что сделает очевидным их применение и позволит избежать дублирования их имен локальными объектами данных.

8.18.1. Атрибут EXTERNAL

Задание атрибута EXTERNAL может быть выполнено как в отдельном операторе, так и в операторе описания типа. Последнее возможно, если мы имеем дело с процедурой-функцией.

EXTERNAL *name* [, *name*] ...

type-spec, EXTERNAL [, *attrs*] :: *name* [, *name*] ...

type-spec - любой оператор объявления типа.

name - имя внешней процедуры. Не может быть именем операторной функции.

Имена внешних процедур с атрибутом EXTERNAL могут быть использованы в качестве параметров других процедур в той программной единице, в которой распространяется действие атрибута. Если у передаваемой процедуры есть родовое имя, то передаваться должно ее специфическое имя. Внутренние процедуры не допускаются в качестве параметров.

Также атрибут EXTERNAL применяется при замене встроенной функции на пользовательскую функцию с тем же именем (разд. 8.12.2). Если в некоторой программной единице имя объекта имеет атрибут EXTERNAL и совпадает с именем встроенной процедуры, то такая встроенная процедура в этой программной единице недоступна.

Нельзя задать атрибут EXTERNAL функции с атрибутом TARGET.

Процедура неявно обладает атрибутом EXTERNAL, если к ней явно задан интерфейс. При этом явное задание атрибута EXTERNAL к этой процедуре недопустимо. Следовательно, в программной единице, содержащей интерфейс к внешней процедуре, эта процедура может быть использована в качестве фактического параметра. Модульная процедура также может быть использована в качестве фактического параметра. Но так как модульные процедуры имеют явно заданный интерфейс, то их имена не должны появляться в операторе EXTERNAL.

Пример. Написать функцию поиска корня уравнения $x = f(x)$ с заданной точностью *eps* на отрезке [a, b] методом простых итераций. Начальное

приближение $x0 = (a + b)/2$. Используя эту функцию, найти на отрезке [0, 3] с точностью *eps* = 0.0001 корни уравнений

$$x = 1/(1{,}2arctgx + \sqrt{x+1}) \qquad \text{(ответ: } x = 0.5435\text{)}$$

и

$$x = (e^{-x} - \sqrt{e^x} + 3{,}7)/3 \qquad \text{(ответ: } x = 0.8614\text{)}.$$

Алгоритм:

1°. Начало.
2°. Задать начальное приближение $x0$, приняв, например, $x0 = (a + b)/2$.
3°. Положить $x = f(x0)$.
4°. Пока $|x - x0| > eps$, выполнить:
    $x0 = x$
    $x = f(x0)$
    конец цикла 4°.
5°. Принять в качестве решения последнее значение переменной x.
6°. Конец.

Проиллюстрируем метод простых итераций на рис. 8.1.

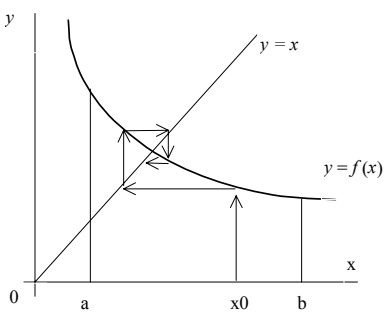


Рис. 8.1. Метод простых итераций

Условия сходимости метода простых итераций: $|f'(x)| < 1$ и $f'(x) < 0$.

Текст программы нахождения корней заданных функций:

```
real function fx1(x)                ! Функции с исходными уравнениями
 real x
 fx1 = 1.0/(1.2 * atan(x) + sqrt(x + 1.0))
end function fx1

real function fx2(x)
 real x
 fx2 = (exp(-x) - sqrt(exp(x)) + 3.7) / 3.0
end function fx2

! Процедура поиска корня уравнения x = f(x)
real function root(fx, a, b, eps)
 real :: fx, a, b, eps, x, x0       ! fx - формальная процедура-функция
 integer :: k, itmax = 100          ! itmax - предельно допустимое число итераций
```

```
 x0 = (a + b)/2.0
 x = fx(x0)
 k = 0
 do while(abs(x - x0) .gt. eps .and. k < itmax)
  k = k + 1
  x0 = x
  x = fx(x0)
 end do
 root = x
end function root
program firo
! Вариант раздела описаний с интерфейсным блоком
! real root
! interface                    ! Заданные в интерфейсном блоке
!  real function fx1(x)        ! процедуры обладают атрибутом
!   real x                     ! EXTERNAL, и их можно использовать
!  end function fx1            ! в качестве параметров процедур
!  real function fx2(x)
!   real x
!  end function fx2
! end interface
! Вариант задания атрибута EXTERNAL в операторе описания
 real, external :: fx1, fx2, root
 write(*, *) 'Корень функции fx1: ', root(fx1, 0.0, 2.0, 1.0e-4)
 write(*, *) 'Корень функции fx2: ', root(fx2, 0.0, 2.0, 1.0e-4)
end program firo
```

Замечание. Использованный критерий останова $|x_n - x_{n-1}| \le \varepsilon$ в общем случае ошибочен и должен быть заменен на $|x_n - x_{n-1}| \le \frac{1-q}{q}\varepsilon$, где $q \ge |f'(x)|$ [7].

8.18.2. Атрибут INTRINSIC

Атрибут INTRINSIC означает, что обладающее им имя является родовым или специфическим именем встроенной процедуры. Родовое имя встроенной процедуры не допускается в качестве фактического параметра, а должно быть использовано ее специфическое имя. Так, недопустимо употреблять в качестве параметра родовое имя функции LOG. Вместо него, например при работе с типом REAL(4), следует описать с атрибутом INTRINSIC имя ALOG и применять затем это специфическое имя в качестве параметра процедуры.

Задание атрибута может быть выполнено как отдельным оператором, так и в операторе описания типа.

INTRINSIC *список имен*

type-spec, INTRINSIC [, *attrs*] :: *список имен*

с*писок имен* - одно или более имен встроенных процедур (в случае нескольких имен они разделяются запятыми). Имя не может одновременно иметь атрибуты INTRINSIC и EXTERNAL. Атрибут INTRINSIC не могут иметь имена определенных пользователем процедур.

С атрибутом INTRINSIC может быть объявлена любая встроенная процедура, однако в качестве фактического параметра процедуры можно использовать только специфические имена приведенных в табл. 8.3 функций. В табл. 8.3 использованы следующие обозначения:

Real для REAL(4) и REAL(8);
Cmp для COMPLEX(4) и COMPLEX(8);
Cmp(4) для COMPLEX(4);
Cmp(8) для COMPLEX(8).

В графе "Типы функций" указаны типы, которые функция имеет, когда она используется в качестве фактического параметра процедуры. Эта информация существенна, когда специфическое и родовое имена функции совпадают. Если родовое имя используется в выражении, то тип функции определяется типом ее параметров.

Таблица 8.3. Специфические имена, которые допускаются в качестве фактических параметров

Описание функции	*Форма вызова с родовым именем*	*Специфические имена*	*Типы аргументов*	*Типы функций*
Абсолютное значение *a*, умноженное на знак *b*	SIGN(*a*, *b*)	ISIGN SIGN DSIGN	Integer Real Real(8)	Integer(4) Real(4) Real(8)
MAX(*x* - *y*, 0)	DIM(*x*, *y*)	IDIM DIM DDIM	Integer Real Real(8)	Integer(4) Real(4) Real(8)
x * *y*	DPROD(*x*, *y*)	DPROD	Real	Real(8)
Усечение	AINT(*a*)	AINT DINT	Real Real(8)	Real(4) Real(8)
Ближайшее целое	ANINT(*a*)	ANINT DNINT	Real Real(8)	Real(4) Real(8)
Ближайшее число типа INTEGER	NINT(*a*)	NINT IDNINT	Real Real(8)	Integer(4) Integer(4)
Абсолютная величина	ABS(*a*)	IABS ABS DABS CABS CDABS	Integer Real Real(8) Cmp(4) Cmp(8)	Integer(4) Real(4) Real(8) Real(4) Real(8)

Остаток по модулю *p*	MOD(*a*, *p*)	MOD AMOD DMOD	Integer Real Real(8)	Integer(4) Real(4) Real(8)
Мнимая часть	AIMAG(*z*)	AIMAG IMAG DIMAG	Cmp Cmp(4) Cmp(8)	Real(4) Real(4) Real(8)
Комплексное сопряжение	CONJG(*z*)	CONJG DCONJG	Cmp(4) Cmp(8)	Cmp(4) Cmp(8)
Квадратный корень	SQRT(*x*)	SQRT DSQRT CSQRT CDSQRT	Real Real(8) Cmp(4) Cmp(8)	Real(4) Real(8) Cmp(4) Cmp(8)
Экспонента	EXP(*x*)	EXP DEXP CEXP CDEXP	Real Real(8) Cmp(4) Cmp(8)	Real(4) Real(8) Cmp(4) Cmp(8)
Натуральный логарифм	LOG(*x*)	ALOG DLOG CLOG CDLOG	Real Real(8) Cmp(4) Cmp(8)	Real(4) Real(8) Cmp(4) Cmp(8)
Десятичный логарифм	LOG10(*x*)	ALOG10 DLOG10	Real Real(8)	Real(4) Real(8)
Синус	SIN(*x*)	SIN DSIN CSIN	Real Real(8) Cmp(4)	Real(4) Real(8) Cmp(4)
Синус (аргумент в град.)	SIND(*x*)	SIND DSIND	Real, Cmp Real(8)	Real(4) Real(8)
Косинус	COS(*x*)	COS DCOS CCOS CDCOS	Real Real(8) Cmp(4) Cmp(8)	Real(4) Real(8) Cmp(4) Cmp(8)
Косинус (аргумент в град.)	COSD(*x*)	COSD DCOSD	Real, Cmp Real(8)	Real(4) Real(8)
Тангенс	TAN(*x*)	TAN DTAN	Real Real(8)	Real(4) Real(8)
Тангенс (аргумент в град.)	TAND(*x*)	TAND DTAND	Real Real(8)	Real(4) Real(8)
Котангенс	COTAN(*x*)	COTAN DCOTAN	Real Real(8)	Real(4) Real(8)
Арксинус	ASIN(*x*)	ASIN DASIN	Real Real(8)	Real(4) Real(8)

Арксинус (результат в град.)	ASIND(*x*)	ASIND DASIND	Real Real(8)	Real(4) Real(8)
Арккосинус	ACOS(*x*)	ACOS DACOS	Real Real(8)	Real(4) Real(8)
Арккосинус (результат в град.)	ACOSD(*x*)	ACOSD DACOSD	Real Real(8)	Real(4) Real(8)
Арктангенс	ATAN(*x*)	ATAN DATAN	Real Real(8)	Real(4) Real(8)
Арктангенс (результат в град.)	ATAND(*x*)	ATAND DATAND	Real Real(8)	Real(4) Real(8)
Арктангенс (*y*/*x*)	ATAN2(*y*, *x*)	ATAN2 DATAN2	Real Real(8)	Real(4) Real(8)
Арктангенс (*y*/*x*) (результат в град.)	ATAN2D(*y*, *x*)	ATAN2D DATAN2D	Real Real(8)	Real(4) Real(8)
Гиперболический синус	SINH(*x*)	SINH DSINH	Real Real(8)	Real(4) Real(8)
Гиперболический косинус	COSH(*x*)	COSH DCOSH	Real Real(8)	Real(4) Real(8)
Гиперболический тангенс	TANH(*x*)	TANH DTANH	Real Real(8)	Real(4) Real(8)
Текстовая длина	LEN(*string*)	LEN	Character	Integer(4)
Начальная позиция	INDEX(*s*, *sub*)	INDEX	"	"

В качестве параметров, даже после их объявления с атрибутом INTRINSIC, не могут быть использованы родовые имена встроенных процедур, а также специфические имена приведенных в табл. 8.4 встроенных процедур.

Таблица 8.4. Специфические имена, не допускаемые в качестве фактических параметров

Описание функции	*Форма вызова с родовым именем*	*Специфические имена*	*Типы аргуметов*	*Типы функций*
Преобразование в целый тип	INT(*a*)	INT IFIX IDINT	Real, Cmp Real(4) Real(8)	Integer Integer(4) Integer
Преобразование в вещественный тип	REAL(*a*)	REAL FLOAT SNGL DREAL	Integer Integer Real(8) Cmp(8)	Real Real Real Real(8)
Мнимая часть	AIMAG(*z*)	IMAG	Cmp(4)	Real(4)
MAX(*a*1, *a*2, ...)	MAX(*a*1, *a*2, ...)	MAX0	Integer	Integer

		AMAX1 DMAX1 AMAX0 MAX1	Real Real(8) Integer Real	Real Real(8) Real Integer
MIN(*a*1, *a*2, ...)	MIN(*a*1, *a*2, ...)	MIN0 AMIN1 DMIN1 AMIN0 MIN1	Integer Real Real(8) Integer Real	Integer Real Real(8) Real Integer

Пример. Построить графики функций sin*x* и cos*x* на отрезке [-π, π].

Для работы в графическом режиме необходимо создать проект как приложение QuickWin или Standard Graphics. Для доступа к процедурам графической библиотеки выполняется ссылка на модуль MSFLIB.

В графическом режиме физическая система координат видового окна начинается в его верхнем левом углу. Для построения графика используем оконную систему координат, расположив начало системы координат в центре окна. Оконная система координат позволяет выполнять графические построения, оперируя реальными координатами. Размеры окна вывода по осям *x* и *y* установим равными половине соответствующих размеров видеоокна. Для определения последних воспользуемся функцией GET-WINDOWCONFIG. Назначение использованных графических процедур можно понять из размещенного в тексте программы комментария. Их подробное описание дано в [1].

```
use msflib
intrinsic dsin, dcos                ! Используем специфические
logical res                         ! имена встроенных функций
integer(2) status2, XE, YE          ! XE,YE - размеры экрана в пикселях
real(8) dx                          ! Используем двойную точность
logical(2) finv /.true./            ! Ось y направлена снизу вверх
real(8), parameter :: pi = 3.14159265
type(windowconfig) wc
! Автоматическая настройка конфигурации окна
data  wc.numxpixels, wc.numypixels, wc.numtextcols, &
      wc.numtextrows, wc.numcolors, wc.fontsize / 6*-1 /
wc.title = "Встроенные функции как параметры процедуры"C
res = setwindowconfig(wc)
res = getwindowconfig(wc)           ! Читаем параметры видеоокна
XE = wc.numxpixels                  ! numxpixels - число пикселей по оси x
YE = wc.numypixels                  ! numypixels - число пикселей по оси y
call axis( )                        ! Рисуем оси координат
! Задание видового порта размером XE/2 * YE/2 в центре видеоокна
call setviewport(XE/4_2, YE/4_2, 3_2*XE/4_2, 3_2*YE/4_2)
! Оконная система координат (ОСК)
status2 = setwindow(finv, -pi, -1.0_8, pi, 1.0_8)
```

```
dx = pi / dble(XE/2)              ! Шаг по оси x
call curve(dsin, dx, 10_2)        ! Рисуем sinx светло-зеленым цветом
call curve(dcos, dx, 14_2)        ! Рисуем cosx желтым цветом
 contains
  subroutine axis( )              ! Рисуем оси координат
   type(xycoord) xy
   status2  = setcolor(15_2)      ! Оси координат - белым цветом
   call moveto(int2(XE/4 - 10), int2(YE/2), xy)
   status2 = lineto(3_2*XE/4_2 + 10_2, YE/2_2)    ! Ось x
   call moveto(int2(XE/2), int2(YE/4 - 10), xy)
   status2 = lineto(XE/2_2, 3_2*YE/4_2 + 10_2)    ! Ось y
  end subroutine axis
  subroutine curve(fx, dx, color)  ! График функции y = fx(x)
   real(8) fx, dx, x, y            ! fx - формальная функция
   integer(2) color
   status2 = setcolor(color)       ! График функции цветом color
   do x = -pi, pi, dx              ! Изменение x в ОСК
    y = fx(x)                      ! Значение y в ОСК
    status2 = setpixel_w(x, y)     ! Вывод точки графика
   end do
  end subroutine curve
end                               ! Результат приведен на рис. 8.2
```

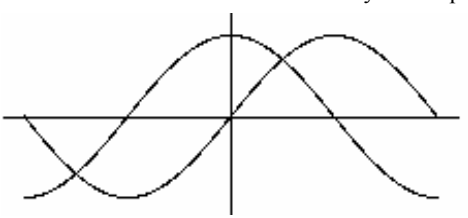


Рис. 8.2. Графики функций sin*x* *и* cos*x*

8.19. Оператор RETURN выхода из процедуры

Выход из процедуры осуществляется в результате выполнения оператора END или оператора RETURN.

Пример. Составить функцию поиска первого отрицательного числа в массиве.

```
real b(20) /1.1, 1.2, -1.3, 1.4, 16*0.0/, bneg, fineg
bneg = fineg(b, 20)               ! Функция fineg возвращает 0, если
if(bneg .eq. 0 ) then             ! в массиве нет отрицательных чисел
  write(*, *) ' В массиве нет отрицательных чисел'
else
  write(*, *) ' Первое отрицательное число', bneg
end if
end
```

```
function fineg (b, n)
 integer  i, n
 real fineg, b(n)
 fineg = 0                    ! Вернем 0, если нет отрицательных чисел
 do i = 1, n
  fineg = b(i)
  if(fineg .lt. 0) return     ! Выход из функции fineg
 end do
end
```

В подпрограммах оператор RETURN может также иметь вид:

RETURN *номер метки*

номер метки - номер звездочки в списке формальных параметров подпрограммы. Такой возврат из подпрограммы называется *альтернативным* и обеспечивает в вызывающей программной единице передачу управления на оператор, метка которого является фактическим параметром и соответствует формальному параметру - звездочке, номер которой указан в операторе RETURN.

Пример альтернативного возврата:

```
integer a(5) /-1, 2, 3, 4, 5/, n /5/
call alre(a, n, *10, *20)    ! Перед меткой обязательна *
write(*, *) ' = 0'           ! На данном наборе данных
go to 40                     ! будет выполнен переход на метку 20
10 write(*, *) ' < 0'
go to 40
20 write(*, *) ' > 0'
40 end

subroutine alre(a, n, *, *)
 integer a(n), sv
 sv = sum(a)
 if(sv .eq. 0) return        ! Нормальный возврат
 if(sv .lt. 0) return 1      ! Передача управления на метку 10
 return 2                    ! sv > 0; передача управления на метку 20
end
```

Замечание. Программы с альтернативным возвратом обладают плохой структурой. Отказаться от альтернативного возврата позволяют конструкции IF и SELECT CASE.

8.20. Оператор ENTRY дополнительного входа в процедуру

Оператор RETURN позволяет организовать несколько точек выхода из процедуры. Наряду с этим в Фортране можно организовать и

дополнительные точки входа во внешнюю или модульную процедуру. Для этого используется оператор ENTRY.

ENTRY *ename* [([*список формальных параметров*])] &
[RESULT (*имя результата*)]

Каждая точка входа задает отдельную процедуру со своим именем *ename*, называемым *именем входа*. Формальные параметры процедуры определяются *списком формальных параметров* оператора ENTRY. Имя точки входа является глобальным и не должно совпадать с другим глобальным именем. Оно также не должно совпадать с локальными именами процедуры, в которой эта точка входа существует.

Предложение RESULT имеет тот же смысл, что и в операторе FUNCTION. *Имя результата* не может совпадать с *ename*.

Вызов подпрограммы с использованием дополнительного входа:

CALL *ename* [([*список фактических параметров*])]

Обращение к функции с использованием дополнительного входа:

result = *ename*([*список фактических параметров*])

В случае функции использование круглых скобок даже при отсутствии фактических параметров обязательно.

При таком вызове выполнение процедуры начинается с первого исполняемого оператора, следующего за оператором ENTRY.

В подпрограмме оператор ENTRY определяет дополнительную подпрограмму с именем *ename*.

В функции оператор ENTRY определяет дополнительную функцию с результирующей переменной *имя результата* или *ename*, если предложение RESULT опущено. Описание результирующей переменной определяет характеристики возвращаемого функцией результата. Если характеристики результата функции, определяемой оператором ENTRY, такие же, как и у главного входа, то обе результирующие переменные (даже если они имеют разное имя) являются одной и той же переменной. В противном случае они ассоциируются в памяти и на них накладываются ограничения: все результирующие переменные должны иметь один вид памяти (текстовая или числовая), должны быть скалярами и не должны иметь атрибут POINTER. В случае текстового результата результирующие переменные должны быть одной длины.

При работе с оператором ENTRY следует соблюдать такие правила:

- внутри подпрограммы *имя входа* не может совпадать с именем формального параметра в операторах FUNCTION, SUBROUTINE или EXTERNAL;
- внутри функции *имя входа* не может появляться ни в одном из операторов функции, кроме оператора объявления типа, до тех пор, пока *имя входа* не будет определено в операторе ENTRY;

- если ENTRY определяет функцию символьного типа, то имена всех точек входа должны быть символьного типа и иметь одну длину;
- формальный параметр оператора ENTRY не может появляться в выполняемом операторе, расположенном до оператора ENTRY. Однако это правило не распространяется на формальный параметр, если он также присутствует в операторах FUNCTION, SUBROUTINE или ранее размещенном операторе ENTRY;
- оператор ENTRY может появляться только во внешней или модульной процедуре;
- оператор ENTRY не может появляться внутри конструкций IF (между IF и END IF), SELECT CASE, WHERE, внутри DO- и DO WHILE-циклов и в интерфейсном блоке;
- нельзя определить точку входа с префиксом RECURSIVE. Задание RECURSIVE в главном входе (в операторах FUNCTION или SUBROUTINE) означает, что заданная точкой входа процедура может обращаться сама к себе.

Интерфейс к процедуре, определяемой точкой входа, если он необходим, задается в самостоятельном теле интерфейсного блока, в заголовке которого должны стоять операторы SUBROUTINE или FUNCTION (а не ENTRY).

Число дополнительных входов в процедуру не ограничено.

Пример. Подпрограмма *vsign* выведет сообщение '>= 0', если *num* ≥ 0, и сообщение '< 0', если *num* < 0.

```
write(*,'(1x, a \)') 'Enter num (INTEGER):    ! Вывод без продвижения
read(*, *) num
if(num .ge. 0) then
 call vsign
else
 call negative
end if
end
subroutine vsign                              ! Главный вход
 write(*, *) '>= 0'
 return
entry negative                                ! Точка входа negative
 write(*, *) '< 0'
 return
end
```

Замечание. Так же как и в случае альтернативного возврата, применение дополнительных входов ухудшает структуру программы и поэтому не может быть рекомендовано для использования.

8.21. Атрибут AUTOMATIC

В CVF и FPS по умолчанию переменные являются в большинстве случаев *статическими*, т. е. под них всегда выделена память и они размещены в памяти статически (адрес размещения статической переменной не меняется в процессе выполнения программы). Переменная называется *автоматической*, если память под эту переменную выделяется по необходимости. Размещение автоматических переменных выполняется в стеке. Примерами автоматических объектов являются объявляемые в процедуре автоматические массивы и строки. Однако в процедуре и модуле можно сделать переменную автоматической, присвоив ей атрибут AUTOMATIC. Атрибут AUTOMATIC является расширением над стандартом Фортрана и может быть задан в отдельном операторе и при объявлении типа:

AUTOMATIC [*список имен переменных*]

type-spec, AUTOMATIC [, *атрибуты*] :: *список имен переменных*

Автоматические переменные прекращают существование после выполнения операторов RETURN или END. Таким образом, их значения при следующем вызове процедуры могут отличаться от тех, которые они получили ранее.

Если оператор AUTOMATIC задан без *списка имен*, то все переменные внутри блока видимости, которые могут иметь атрибут AUTOMATIC, будут неявно объявлены автоматическими.

В операторе AUTOMATIC не могут появляться:

- имена и объекты *common*-блоков;
- переменные, имеющие атрибут SAVE;
- переменные с атрибутами ALLOCATABLE или EXTERNAL;
- формальные параметры и имена процедур.

Переменная, которой явно присвоен атрибут AUTOMATIC, не может быть инициализирована в операторе DATA или в операторе объявления типа. Переменные, неявно ставшие автоматическими и появившиеся в операторе DATA или инициализированные в операторе объявления типа, получат атрибут SAVE и будут помещены в статическую память.

Переменная не может появляться в операторе AUTOMATIC более одного раза.

Пример:

```
call atav(2, 3)
call atav(4, 5)
end
subroutine atav (m, n)
 automatic                    ! Переменные объявляются автоматическими неявно
```

```
 integer :: m, n, a, b = 2
 print *, 'b = ', b              ! Переменная b является статической,
 a = m                           ! поэтому сохраняет полученное значение
 b = n                           ! при повторном вызове
end                              ! Переменная a является автоматической
```

Результат:

```
b = 2
b = 3
```

8.22. Атрибут SAVE

Существующая в процедуре или модуле переменная будет сохранять свое значение, статус определенности, статус ассоциирования (для ссылок) и статус размещения (в случае размещаемых массивов) после выполнения оператора RETURN или END, если она имеет атрибут SAVE. В CVF и FPS все переменные (кроме объектов с атрибутом ALLOCATABLE или POINTER и автоматических объектов) по умолчанию имеют такой атрибут. Поэтому объявление переменной с атрибутом SAVE выполняется для создания программ, переносимых на другие платформы или в том случае, если переменные процедуры или модуля неявно получили атрибут AUTOMATIC. Атрибут SAVE задается отдельным оператором или при объявлении типа:

SAVE [[::] *список объектов*]

type-spec, SAVE [, *атрибуты*] :: *список объектов*

Список объектов может включать имена переменных и *common*-блоков. Последние при задании обрамляются слешами. Один и тот же объект не может дважды появляться в операторе SAVE.

Если оператор SAVE задан без *списка объектов*, то все объекты программной единицы, которые могут иметь атрибут SAVE, получают этот атрибут.

Задание атрибута SAVE в главной программе не имеет никакого действия. Если *common*-блок задан в главной программе, то он и, следовательно, все его переменные имеют атрибут SAVE. Если же *common*-блок задан только в процедурах, то он должен быть сохранен в каждой использующей его процедуре.

Атрибут SAVE не может быть задан:

- переменным, помещенным в *common*-блок;
- формальным параметрам процедур;
- именам процедур и результирующей переменной функции;
- автоматическим массивам и строкам (разд. 4.8.3);
- объектам, явно получившим атрибут AUTOMATIC.

Пример:

```
subroutine shosa( )
 real da, a, dum
 common /bz/ da, a, dum(10)
 real(8), save :: x, y
 save /bz/
```

8.23. Атрибут STATIC

Имеющие атрибут STATIC переменные (в процедуре или модуле) сохраняются в (статической) памяти в течение всего времени выполнения программы. Атрибут является расширением над стандартом Фортрана и эквивалентен ранее приведенному атрибуту SAVE и атрибуту STATIC языка СИ. Значения статических переменных сохраняются после выполнения оператора RETURN или END. Напомним, что в CVF и FPS по умолчанию переменные (кроме динамических) размещены в статической памяти. Для изменения правил умолчания используются атрибуты ALLOCATABLE, AUTOMATIC и POINTER. Атрибут STATIC может быть задан в двух формах:

STATIC [[::] *список объектов*]

type-spec, STATIC [, *атрибуты*] :: *список объектов*

Список объектов может включать имена переменных и *common*-блоков. Имена последних при включении их в *список объектов* оператора STATIC обрамляются слешами.

Пример:

```
integer :: ng = -1
do while(ng /= 0)                    ! Цикл завершается при ng = 0
 call sub1(ng, ng + ng)
 print *, 'Enter integer non zero value to continue or zero to quit'
 read *, ng
end do

 contains

  subroutine sub1(iold, inew)
   integer, intent(inout) :: iold
   integer, static :: n2             ! При каждом вызове n2 = -1
   integer, automatic :: n3
   integer, intent(in) :: inew
   if(iold == -1) then
    n2 = iold
    n3 = iold                        ! Значение n3 определено только при ng = -1
   end if
   print *, 'new: ', inew, '  n2: ', n2, '  n3: ', n3
  end subroutine
end
```

8.24. Атрибут VOLATILE

Атрибут может быть задан только в CVF и является расширением над стандартом Фортрана. Атрибут указывает компилятору, что значение объекта непредсказуемо. Объект, обладающий атрибутом VOLATILE, не будет оптимизироваться в процесс компиляции. Как и другие, атрибут может быть задан в двух формах:

VOLATILE *список объектов*

type-spec, VOLATILE [, *атрибуты*] :: *список объектов*

Список объектов может включать имена переменных и *common*-блоков. Имена последних при включении их в *список объектов* оператора VOLATILE обрамляются слешами.

Переменная или *common*-блок должны объявляться VOLATILE, если способ их определения неочевиден для компилятора. Например, если операционная система размещает переменную в разделяемой памяти, с тем чтобы ее могла использовать другая программа, которая может в том числе и изменить значение переменной, или в случае ассоциирования памяти посредством оператора EQUIVALENCE.

Если составной объект (массив, производный тип) объявляется VOLATILE, то каждый его элемент получает этот атрибут. Аналогично если *common*-блок объявляется VOLATILE, то этим атрибутом обладает каждый его элемент.

Атрибут VOLATILE не может быть задан процедуре, результату функции и *namelist*-группе.

Пример:

```
logical(kind = 1) ipi(4)
integer(kind = 4) a, b, c, d, e, ilook
integer(kind = 4) p1, p2, p3, p4
common /blk1/ a, b, c
volatile /blk1/, d, e
equivalence(ilook, ipi)
equivalence(a, p1)
equivalence(p1, p4)
```

Именованный *common*-блок *blk*1, переменные *d* и *e* объявляются VOLATILE явно. Поведение переменных *p*1 и *p*4 в результате их ассоциирования по памяти (прямого и непрямого) с *volatile*-переменной *a* зависит от *a*.

8.25. Чистые процедуры

Чистыми называются процедуры, не имеющие побочных эффектов. Пример побочного эффекта демонстрирует следующая программа:

```
program side_effect
 real(4) :: dist, d, p = 3.0, q = 4.0, r = 5.0
```

```
 d = max(dist(p, q), dist(q, r))
 print *, d                          !      7.071068
end program side_effect

function dist(p, q)
 real(4) :: dist, p, q
 dist = sqrt(p * p + q * q)
 q = dist                            ! Изменение q - побочный эффект
end function dist
```

Суть его в том, что функция *dist* переопределяет значение параметра *q*. А это означает, что второй вызов функции *dist* при вычислении *d*, выполняется при *q*, равном 5.0, вместо ожидаемого первоначального значения *q* = 4.0. Такие эффекты запрещены стандартом и должны отслеживаться и устраняться программистом.

Сообщение о том, что процедура является чистой, обеспечивается ключевым словом PURE, применяемым в заголовке процедуры:

[*type-spec*] PURE SUBROUTINE | FUNCTION *name* &
[RESULT (*resultname*)]

или

PURE [*type-spec*] SUBROUTINE | FUNCTION *name* &
[RESULT (*resultname*)]

type-spec - тип результирующей переменной функции.
name - имя процедуры.
resultname - имя результирующей переменной функции.

Чистая процедура характеризуется тем, что:

- функция возвращает значение и не меняет ни одного из своих параметров;
- подпрограмма изменяет только те параметры, которые имеют вид связи INTENT(OUT) и INTENT(INOUT).

По умолчанию чистыми являются:

- все встроенные функции и встроенная подпрограмма MVBITS;
- процедуры библиотеки высокоскоростного Фортрана, применяемого для параллельных вычислений под Юниксом.

В чистых процедурах все формальные параметры, кроме формальных процедур и ссылок, должны иметь вид связи:

- для функций - только INTENT(IN);
- для подпрограмм - любой: INTENT(IN, или OUT, или INOUT).

Никакие локальные переменные чистой процедуры, в том числе и относящиеся к внутренним процедурам, не должны:

- обладать атрибутом SAVE;
- быть инициализированными в операторах объявления или DATA.

В чистых процедурах имеются ограничения на использование:

- глобальных переменных;
- формальных параметров с видом связи INTENT(IN) или с необъявленным видом связи;
- объектов, ассоциируемых по памяти с какими-либо глобальными переменными.

Ограничения таковы: перечисленные объекты не должны использоваться:

1) в случаях, когда возможно изменение их значения. Это может произойти, если переменная является:
- левой частью оператора присваивания или прикрепления ссылки (если объект является ссылкой);
- фактическим параметром, ассоциированным с формальным параметром с видом связи INTENT(OUT или INOUT) или обладающим атрибутом POINTER;
- индексной переменной операторов DO, FORALL или встроенного DO-цикла;
- переменной оператора ASSIGN;
- элементом списка ввода оператора READ;
- именем внутреннего файла оператора WRITE;
- объектом операторов ALLOCATE, DEALLOCATE или NULLIFY;
- спецификатором IOSTAT или SIZE операторов B/B или STAT операторов ALLOCATE и DEALLOCATE;

2) в создании ссылки, например в качестве адресата или в качестве элемента правой части оператора присваивания переменной производного типа, если он имеет ссылочный компонент на любом из его уровней.

Чистые процедуры не должны содержать:

- операторы B/B во внешние файлы или устройства;
- операторы PAUSE и STOP.

Чистые процедуры предназначены для вызова в тех случаях, когда вызов иных, не владеющих ключевым словом PURE процедур недопустим:

- в операторе FORALL или его выражении-маске;
- из другой чистой процедуры.

Также только чистую процедуру можно использовать в качестве параметра другой чистой процедуры.

Если чистая процедура используется в приведенных ситуациях, то ее интерфейс должен быть задан явно и она должна быть объявлена в нем с ключевым словом PURE. Напомним, что все встроенные процедуры являются чистыми и по умолчанию обладают явным интерфейсом.

Пример:

```
pure function decr(k, m)
 real(4) :: decr
```

```
 integer(4), intent(in) :: k, m        ! Формальные параметры чистой функции
 decr = real(m) / real(k)              ! должны иметь вид связи INTENT(IN)
end function decr
program pudem
 real(4), dimension(5, 5) :: array = 5.0
 interface
  pure function decr(k, m)             ! Поскольку функция используется в FORALL,
   real(4) :: decr                     ! то необходимо задать ее интерфейс
   integer(4), intent(in) :: k, m
  end function decr
 end interface
 forall(i = 1:5, j = 1:5) array(i, j) = decr(i, j)
 print '(10f5.1)', array(1, :)         !  1.0  2.0  3.0  4.0  5.0
end program pudem
```

Замечание. Чистые процедуры введены стандартом 1995 г.

8.26. Элементные процедуры

Элементные пользовательские процедуры подобно встроенным элементным процедурам могут иметь в качестве фактических параметров либо скаляры, либо массивы. В последнем случае массивы должны быть согласованы, т. е. иметь одинаковую форму; результатом процедуры является поэлементная обработка массивов - фактических параметров. Приведем пример выполнения встроенной элементной функции MOD, возвращающей остаток от деления первого параметра на второй:

```
integer(4), dimension(5) :: a = (/ 1, 2, 3, 4, 5 /), b = (/ 1, 2, -2, 4, 3 /), c
integer(4) :: d
c = mod(a, b)                          ! Параметры функции - массивы
print *, c                             !  0  0  1  0  2
d = mod(b(4), a(3))                    ! Параметры функции - скаляры
print *, d                             !  1
```

Программная единица, вызывающая элементную функцию, должна содержать ее интерфейс, в котором явно указано слово ELEMENTAL.

Цель введения элементных функций - упростить распараллеливание вычислений на многопроцессорных машинах: компилятор, имеющий сведения о том, что функция элементная, выполняет распараллеливание по заложенным в него правилам.

Элементные функции - это чистые функции, имеющие только скалярные формальные параметры, не являющиеся ссылками или процедурами. Вид связи параметров - INTENT(IN). Результирующая переменная элементной функции также является скаляром и не может быть ссылкой. Элементная функция снабжается ключевым словом ELEMENTAL, которое автоматически подразумевает ключевое слово PURE. Элементные функции не могут быть оснащены ключевым словом RECURSIVE.

Если фактическими параметрами элементной функции являются массивы, то они должны быть согласованы; результатом такой функции является массив, согласованный с массивами-параметрами.

Пример:

```
elemental integer(4) function find_c(a, b)
 integer(4), intent(in) :: a, b        ! Не забываем задать вид связи INTENT(IN)
 if(a > b) then
   find_c = a
 else if(b < 0) then
   find_c = abs(b)
 else
   find_c = 0
 end if
end function find_c

program etest
 interface                             ! Интерфейс обязателен
  elemental integer(4) function find_c(a, b)
    integer(4), intent(in) :: a, b     ! Обязательное задание вида связи INTENT(IN)
  end function find_c
 end interface
 integer(4), dimension(5) :: a = (/ -1, 2, -3, 4, 5 /), b = (/ 1, 2, -2, 4, 3 /), c
 integer(4) :: d = 5
 c = find_c(a, b)                      ! Параметры функции - массивы
 print *, c                            !    0    0    2    0    5
 d = find_c(-1, 1)                     ! Параметры функции - скаляры
 print *, d                            !    0
end program etest
```

Замечание. Поскольку элементные функции являются чистыми, они могут быть использованы в операторе и конструкции FORALL.

Элементные подпрограммы задаются подобно элементным функциям. В теле процедуры могут изменяться параметры с видом связи OUT и INOUT.

Пример:

```
elemental subroutine find_c(a, b, c)
 integer(4), intent(in) :: a, b
 integer(4), intent(out) :: c
 if(a > b) then
  c = a
 else if(b < 0) then
  c = abs(b)
 else
  c = 0
```

```
 end if
end subroutine find_c
program etest2
 interface
  elemental subroutine find_c(a, b, c)
    integer(4), intent(in) :: a, b
    integer(4), intent(out) :: c
  end subroutine
 end interface
 integer(4), dimension(5) :: a = (/ -1, 2, -3, 4, 5 /), b = (/ 1, 2, -2, 4, 3 /), c
 integer(4) :: d = 5
```

```
call find_c(a, b, c)                ! Параметры и результат - массивы
 print *, c                         !   0    0    2    0    5
 call find_c(-1, 1, d)              ! Параметры и результат - скаляры
 print *, d                         !   0
end program etest2
```

Замечание. Элементные процедуры введены стандартом 1995 г.

8.27. Операторные функции

Если некоторое выражение встречается в программной единице неоднократно, то его можно оформить в виде операторной функции и заменить все вхождения выражения на эту функцию. Операторные функции задаются так:

имя функции ([*список формальных параметров*]) = *выражение*

Если *список формальных параметров* содержит более одного имени, то имена разделяются запятыми.

Как и встроенная или внешняя функция, операторная функция вызывается в выражении. Областью видимости операторной функции является программная единица, в которой эта функция определена. В то же время операторная функция может быть доступна в других программных единицах за счет ассоциирования через носитель или *use*-ассоциирования, но не может быть ассоциирована через параметры процедуры. Тип операторной функции следует объявлять явно, размещая ее имя в операторе объявления типа или в операторе IMPLICIT.

Пример. Выполнить табуляцию функции $z = \sin y * e^{-x}$.

```
real(8) :: x = -1.0_8, y, z          ! Используем двойную точность
real(8) :: dx = 0.4_8, dy = 0.3_8
z(x, y) = exp(-x) * sin(y)           ! Задание операторной функции z(x, y)
write(*, '(6h  x/y , 20f8.2)') (y, y = -0.6, 0.6, 0.3)
do while(x <= 1.0_8)
 write(*, '(f6.2 \)') x              ! Вывод x без перехода на новую строку
 y = -0.60_8
 do while(y <= 0.6_8)
  write(*, '(f8.2 \)') z(x, y)       ! Вывод z без перехода на новую строку
  y = y + dy
 end do
 x = x + dx
 write(*, *)                         ! Переход на новую строку
end do
end
```

Замечание. Для вывода без продвижения на новую строку используется преобразование обратного слеша (\).

8.28. Строка INCLUDE

В больших программах исходный код целесообразно хранить в разных файлах. Это упрощает работу над фрагментами программы и над программой в целом. Включение исходного кода одного файла в код другого можно выполнить при помощи директивы $INCLUDE или строки INCLUDE, имеющей вид:

INCLUDE '*имя файла*'

имя файла - заключенное в апострофы или двойные кавычки имя текстового файла с исходным кодом фрагмента Фортран-программы. При необходимости *имя файла* должно содержать и путь к файлу.

Строка INCLUDE не является оператором Фортрана. Она вставляет содержимое текстового файла в то место программной единицы, где он расположен. При этом строка INCLUDE замещается вставляемым текстом. Компилятор рассматривает содержимое вставленного файла как часть исходной программы и выполняет компиляцию этой части сразу после ее вставки. После завершения компиляции вставленного файла компилятор продолжает компиляцию исходной программной единицы начиная с оператора, следующего сразу после строки INCLUDE.

Включаемый файл может содержать другие строки INCLUDE, но не должен прямо или косвенно ссылаться сам на себя. Такие включаемые файлы называются *вложенными*. Компилятор позволяет создавать вложенные включаемые файлы, содержащие до 10 уровней вложения с любым набором строк INCLUDE.

Первая строка включаемого файла не должна быть строкой продолжения, а его последняя строка не должна содержать перенос. Перед оператором не может быть поставлена метка.

В Фортране *include*-файлы рассматриваются как избыточное средство языка и могут быть практически полностью и с большим эффектом заменены модулями. Модули не только обеспечивают доступ к расположенным в модуле операторам объявления и описания и размещенным после оператора CONTAINS модульным процедурам, но и позволяют выполнять (за счет *use*-ассоциирования) обмен данными между использующими модули программными единицами.

8.29. Порядок операторов и директив

Операторы и директивы в программных единицах должны появляться в приведенном в табл. 8.5 порядке.

Таблица 8.5. Последовательность операторов и директив

<table>
<tr><td colspan="3">$INTEGER, $REAL, $[NO]SRICT, $OPTIMIZE</td><td>$ATTRIBUTES</td></tr>
<tr><td colspan="3">BLOCK DATA, FUNCTION, MODULE, PROGRAM, SUBROUTINE</td><td rowspan="13">$[NO]DEBUG
$[NO]DECLARE
$DEFINE, $UNDEFINE
$IF, $IF DEFINED
$ELSE, $ELSEIF, $END IF
$FIXFORMLINESIZE
$[NO]FREEFORM
$INCLUDE, $LINE
$LINESIZE, $[NO]LIST
$MESSAGE
$OBJCOMMENT, $PACK
$PAGE, $PAGESIZE
$SUBTITLE, $TITLE</td></tr>
<tr><td colspan="3">USE-операторы</td></tr>
<tr><td>IMPLICIT NONE</td><td rowspan="2">PARAMETER</td><td rowspan="6"></td></tr>
<tr><td>IMPLICIT</td></tr>
<tr><td>Определения производных типов
Интерфейсные блоки
Операторы объявления типа
Операторы объявления
Операторные функции</td><td rowspan="2">PARAMETER
DATA</td><td rowspan="2">ENTRY
FORMAT</td></tr>
<tr><td>Исполняемые операторы</td><td>DATA</td></tr>
<tr><td colspan="3">CONTAINS
Внутренние и модульные процедуры
END</td></tr>
</table>

В табл. 8.6 для разных программных компонентов указаны операторы, которые могут в них появляться. Строка "Объявления" подразумевает операторы PARAMETER, IMPLICIT, объявления типов данных и их атрибутов.

Таблица 8.6. Операторы программных компонентов

Операторы	*Главная программа*	*Модуль*	*BLOCK DATA*	*Внешняя процедура*	*Модульная процедура*	*Внутренняя процедура*	*Тело интерфейса*
USE	Да	Да	Да	Да	Да	Да	Да
ENTRY	Нет	Нет	Нет	Да	Да	Нет	Нет
FORMAT	Да	Нет	Нет	Да	Да	Да	Нет
Объявления	Да	Да	Да	Да	Да	Да	Да
DATA	Да	Да	Да	Да	Да	Да	Нет
Определения производных типов	Да	Да	Да	Да	Да	Да	Да
Интерфейсные блоки	Да	Да	Нет	Да	Да	Да	Да
Операторные функции	Да	Нет	Нет	Да	Да	Да	Нет
Исполняемые операторы	Да	Нет	Нет	Да	Да	Да	Нет
CONTAINS	Да	Да	Нет	Да	Да	Нет	Нет

9. Форматный ввод/вывод

Данные в памяти ЭВМ хранятся в двоичной форме, представляя собой последовательность нулей и единиц. С особенностями представления различных типов данных в ЭВМ можно познакомиться, например, в [5]. Употребляемые в Фортране модели данных целого и вещественного типа рассмотрены в разд. 6.11.1.

Входные и выходные данные часто необходимо представить в ином, отличном от внутреннего представления виде. Тогда и возникает задача преобразования данных из входной формы в машинное (внутреннее) представление и, наоборот, из машинного представления во внешнее, например текстовое или графическое.

Стандартные средства Фортрана поддерживают 4 вида В/В данных:

- форматный;
- под управлением списка В/В;
- неформатный;
- двоичный.

Первые два вида предназначены для преобразования текстовой информации во внутреннее представление при вводе и, наоборот, из внутреннего представления в текстовое при выводе. Выполняемые преобразования при форматном В/В задаются списком дескрипторов преобразований. Управляемый списком В/В по существу является разновидностью форматного В/В: преобразования выполняются по встроенным в Фортран правилам в соответствии с типами и значениями элементов списка В/В. Управляющий передачей данных список может быть именованным или неименованным.

В настоящей главе мы рассмотрим только два первых вида передачи данных: форматный и под управлением списка. Неформатный и двоичный В/В рассмотрены в гл. 10.

9.1. Преобразование данных. Оператор FORMAT

Перевод данных из внутреннего представления в текстовое задается *дескрипторами преобразований* (ДП). Так, для вывода *вещественного* числа на поле длиной в 8 символов, в котором 3 символа отведены для представления дробной части, используется дескриптор F8.3. Максимальное значение, которое можно отобразить на заданном поле, равно 9999.999, а минимальное - -999.999. Для преобразования внутреннего представления *целого* числа в текст длиной в 10 символов применяется дескриптор I10. Чтобы напечатать символьную переменную в поле длиной 25 знаков, применяется преобразование A25.

Дескрипторы преобразования содержатся в спецификации формата, например:

```
real :: a = -345.456
integer :: k = 32789
character(20) :: st = 'Строка вывода'
write(*, '(1x, f8.3)') a              !-345.456
write(*, '(1x, i10)') k               !□□□□□32789
write(*, '(1x, a25)') st              !□□□□□□□□□□□□Строка вывода
```

Замечание. Символ □ использован для обозначения пробела.

Спецификация формата включает заключенный в скобки список ДП. Спецификация может быть задана как встроенная в оператор В/В символьная строка, например:

```
'(F8.3, I10)'
```

или как отдельный оператор FORMAT, на который операторы В/В ссылаются при помощи метки. Общий вид оператора:

метка FORMAT (*список* ДП)

ДП разделяются в *списке* ДП запятыми. Например:

```
write(*, '(1x, f8.3, i10)') a, k      !-345.456□□□□□32789
write(*, 1) a, k                      !-345.456□□□□□32789
1 format(1x, f8.3, i10)
```

В каждой из приведенных спецификаций формата содержится ДП 1X. При форматном выводе его присутствие необходимо, правда, только в FPD. Дело в том, что в FPS при форматном выводе по умолчанию первая позиция строки вывода предназначена для простановки символа управления кареткой печатающего устройства. Возможные символы управления кареткой приведены в табл. 9.1.

Таблица 9.1. Символы управления кареткой печатающего устройства

Символ	*Действие*
Пробел	Начать новую строку
+	Остаться на той же строке (перепечатать)
0	Пропустить одну строку
1	Перейти на начало следующей страницы

Поэтому первая позиция строки вывода на экране и печатающем устройстве не отображается.

Чтобы исключить генерацию ложных символов управления кареткой в FPS существует два средства:

- можно всегда вставлять при форматном выводе по крайней мере один пробел в качестве первого символа в каждую запись. Это выполняется дескриптором 1X или T2;
- можно подсоединить внешнее устройство, задав в операторе OPEN спецификатор CARRIAGECONTROL = 'LIST' (разд. 11.4). В этом случае первый символ каждой записи при форматном выводе не будет интерпретироваться как символ управления кареткой и будет выводиться на внешнем устройстве, например:

```
write(*, 1) 'abcd'                ! bcd
open(6, carriagecontrol = 'list') ! по умолчанию устройство 6 - это экран монитора
write(*, 1) 'abcd'                ! abcd
1 format(a)
```

В CVF по умолчанию CARRIAGECONTROL = 'LIST', поэтому предварять список ДП дескриптором 1X, если правила умолчания не изменены, нет необходимости.

Пример:

```
write(*, '(i3)') 123              ! CVF:   123
end                               ! FPS:   23
```

Замечание. В DS есть возможность редактирования оператора FORMAT. Для этого установите курсор на оператор FORMAT, в котором есть хотя бы один ДП, и затем выполните цепочку Edit - Fortran Format Editor.

9.2. Программирование спецификации формата

Спецификацией формата является символьная строка. Наиболее часто значение этой строки задается в виде буквальной символьной константы так, как это было выполнено в примерах предыдущего раздела. Однако в общем случае спецификацией формата может быть и символьная переменная, значение которой может изменяться в процессе вычислений.

Пример. Запрограммировать формат для вывода заголовка по центру экрана. Задачу решить, предполагая, что длина заголовка меньше ширины экрана.

Введем обозначения: *tl* - длина заголовка без завершающих пробелов; *sl* - ширина экрана (в текстовом режиме ширина экрана составляет 80 символов). Для центрирования заголовка необходимо отступить от левой границы экрана $n = (sl - tl)/2$ символов, а затем вывести заголовок. Так, при $tl = 60$ следует применить формат

'(11X, A)' или '(T12, A)'

а при выводе заголовка длиной в 40 символов подошел бы формат

'(21X, A)' или '(T22, A)'

Текст программы формирования формата вывода заголовка длиной *tl*:

```
program t2
character(78) :: title = 'Пример заголовка'
character(20) form                  ! Строка формата вывода
integer(1) tl, n, sl /80/
tl = len_trim(title)                ! Длина заголовка без завершающих пробелов
n = (sl - tl) / 2
! Формирование формата вывода (строки fmt) с дескриптором X
write(form, '(a, i2, a)') '(', n, 'x' // ',' // 'a' // ')'
! или в случае использования дескриптора T
! write(form, '(a, i2, a)') '(' // 't', n + 1, ',' // 'a' // ')'
write(*, form) title                ! Вывод заголовка
end program t2
```

Пояснения:

1. Строка является внутренним файлом, при работе с которым используется форматный В/В.

2. Дескриптор T*n* смещает позицию В/В на *n* символов вправо.

Замечания:

1. Задание формата '(*n*X, A)' или '(T*n*, A)' является ошибкой, так как в этом случае дескриптор содержит недопустимый для формата символ *n*, вместо которого должна быть использована буквальная положительная целая константа без знака. Далее, правда, мы покажем, что такие целые константы могут быть заменены заключенным в угловые скобки целочисленным выражением (разд. 9.3).

2. На самом деле при выводе в DOS-окно заголовок, содержащий русский текст, будет выведен как нечитаемый набор символов. Чтобы поправить положение, необходимо использовать приведенную в прил. 1 функцию RuDosWin, принадлежащую модулю TextTransfer, и внести в приведенный выше код следующие изменения:

```
module TextTransfer
 ! Код модуля TextTransfer см. в прил. 1
 ...
end module TextTransfer

program t2
use TextTransfer                            ! Для вывода русского текста
character(78) :: title = 'Пример заголовка'
...                                         ! Код программы t2 см. выше
write(*, form) trim(RuDosWin(title, .false.))   ! Вывод заголовка
end program t2
```

Далее, впрочем, как и ранее, ссылка на модуль TextTransfer будет опускаться, но всегда, если выводится русский текст, будет подразумеваться, так же как и соответствующее употребление RuDosWin.

Формат также может быть задан в виде символьного массива, элементы которого при выполнении В/В конкатенируются.

Пример. Запрограммировать формат вывода заголовка с применением символьного массива.

```
character(78) :: title = 'Пример заголовка'
character(1) fmt(12)/'(', '1', 'x', ',', 't', 2*' ', ',', 'a', ')', 2*' '/
integer(1) n, sl /80/
n = (sl - len_trim(title))/2
select case(n)
case(1:9)                          ! Преобразуем n в символьное
 write(fmt(6), '(i1)') n           ! представление и занесем в fmt(6)
case(10:)                          ! или в fmt(6) и fmt(7), если n > 9
 write(fmt(6), '(i1)') n/10
 write(fmt(7), '(i1)') mod(n, 10)
endselect
write(*, fmt) title
```

Символы строки или элементы массива, расположенные после крайней правой скобки строки - спецификации формата, игнорируются. Поэтому и строка и массив могут содержать большее число элементов, чем необходимо для задания формата.

При программировании формата надо помнить, что спецификация формата должна быть полностью установлена перед началом выполнения оператора В/В. Во время исполнения оператора В/В ни один символ спецификации формата не может быть изменен.

9.3. Выражения в дескрипторах преобразований

Если в строке формата дескриптор преобразований использует целочисленную константу, то она может быть заменена заключенным в угловые скобки (< >) целочисленным выражением:

```
integer :: m, k
k = 10
do m = 3, 5
 k = k*10
 write(*, '(2x, i<m>)') k          ! 100
end do                             ! 1000
end                                ! 10000
```

Целочисленное выражение может быть любым допускаемым выражением со следующими ограничениями:

- в дескрипторе H константа не может быть заменена целочисленным выражением;
- операции отношения в этом выражении не могут быть заданы в графическом виде, например вместо знака > используется .GT., вместо <= - .LE..

Задаваемое вместо константы целочисленное выражение не может появляться в операторе присваивания при программировании строки формат; так, ошибочен фрагмент:

```
integer :: m = 2, k = 5
character(80) s
s = '(2x, i<k - m>)'
write(*, s) m + k                    ! Ошибка
```

Но корректны операторы:

```
integer :: m = 2, k = 5
write(*, '(2x, i<k - m>)') m + k    !  7
write(*, 1) m, k                    !  2  5
print 1, m, k                       !  2  5
1 format(<m>x, <m>i<k - m>)         ! Правильно
```

Заменяя в ДП константу на выражение, нужно следить за тем, чтобы выражение было целочисленным и возвращаемое им значение было больше нуля.

Пример. Вывести заголовок по центру экрана.

```
character(78) :: title = 'Пример заголовка'
integer(1) :: tl, sl = 80            ! sl - ширина экрана
tl = len_trim(title)
write(*, fmt = 10) title             ! Вывод заголовка
10 format(< (sl - tl)/2 > x, a)
```

9.4. Задание формата в операторах ввода/вывода

При форматном В/В операторы В/В содержат ссылку на используемый формат. Такая ссылка может быть задана четырьмя способами:

- в виде метки, указывающей на оператор формата:

  ```
  write(*, 10) a, k
  ```

  или

  ```
  write(*, fmt = 10) a, k
  10 format(1x, f8.3, i10)
  ```

  Замечание. Метка в случае ее использования для ссылки на формат может быть присвоена целочисленной переменной оператором ASSIGN (прил. 4), который, правда, удален из Фортрана стандартом 1995 г.:

  ```
  integer :: label, m = 55
  assign 20 to label
  print label, m                     !  55
  20 format(1x, i5)
  ```

- в виде встроенного в оператор В/В символьного выражения:

  ```
  write(*, '(1x, f8.3, i10)') a, k
  ```

или

```
write(*, fmt = '(1x, f8.3, i10)') a, k
```

- в виде имени именованного списка В/В:

```
integer :: k = 100, iarray(3) = (/ 41, 42, 43 /)
real :: r4*4 = 24.0, r8*8 = 28.0
namelist /mesh/ k, r4, r8, iarray
write(*, mesh)
```

или

```
write(*, nml = mesh)
```

- в виде звездочки, указывающей на использование управляемого неименованным списком В/В:

```
write(*, *) a, k
write(*, fmt = *) a, k
```

9.5. Списки ввода/вывода

Оператор ввода для каждого элемента списка ввода находит во внешнем файле поле с данными и читает из него в элемент списка значение. Оператор вывода создает в файле по одному полю с данными для каждого элемента списка вывода. В случае форматного вывода размер поля определяется примененным форматом.

9.5.1. Элементы списков ввода/вывода

Элементами списка В/В могут быть как полные объекты данных любых типов (скаляры и массивы), так и их подобъекты, например компоненты записи, элементы массива, сечение массива, подстрока. Между списками ввода и вывода есть различия: список ввода может содержать только переменные и их подобъекты, список вывода содержит выражения.

Пример:

```
type point
 real x, y
 character(8) st
end type point
type(point) p(20), pxy
open(1, file = 'a.txt')
read(1, '(2f8.2)') pxy.x, pxy.y              ! Возможные списки ввода
read(1, '(f8.2 / f8.2 / a)') p(1).x, p(1).y, p(1).st
read(1, 20) pxy                              ! В списке ввода 3 элемента
read(1, 20) (p(k), k = 1, 20)                ! В списке ввода 60 элементов
20 format(2f8.2, a)
```

Присутствующий в списке В/В скалярный объект встроенного типа (кроме комплексного) создает один элемент В/В. Массив встроенного типа (кроме комплексного) добавляет в список В/В все свои элементы. Порядок

следования элементов массива в списке В/В совпадает с порядком их расположения в памяти ЭВМ. Так, эквивалентны списки:

```
real a(2, 3) / 1.1, 2.2, 3.3, 4.4, 5.5, 6.6 /
write(*, *) a                              ! В списке вывода 6 элементов
write(*, *) a(1, 1), a(2, 1), a(1, 2), a(2, 2), a(1, 3), a(2, 3)
```

Скаляр комплексного типа создает два элемента В/В. В случае комплексного массива из *n* элементов в список В/В добавляется 2**n* элементов. Компоненты скаляра производного типа располагаются в списке В/В в том же порядке, в котором они располагаются и в операторе объявления этого типа.

В случае форматного ввода число присутствующих во внешнем файле полей ввода должно быть не меньше числа элементов в списке В/В. Размер занимаемого вводимой величиной поля и его положение в файле должны быть согласованы с форматом ввода. Например:

```
integer(2) k, m, a(20), b(10)
complex(4) z
character(30) art(15)
character(30) ::  fmt = '(4i4 / 10i3 / 2f8.2 / (a30))'
open(1, file = 'a.txt')
read(1, fmt) k, m, a(2), a(4), b, z, art
```

Список ввода содержит 31 элемент: 25 из них дают массивы *b* и *art*, 2 - комплексная переменная *z* и по одному переменные *k*, *m*, *a*(2), *a*(4). Следовательно, не менее 31 значения должно присутствовать и в файле, из которого выполняется ввод данных. Число строк в файле не может быть менее 18, поскольку в спецификации формата *fmt* присутствует преобразование слеша (/), которое обеспечивает перемещение файлового указателя на начало новой записи. Положение и состав полей файла может быть, например таким (символ □ использован для обозначения пробела):

```
□111□222□333□444
□11□12□13□14□15□16□17□18□19□20
□1111.11□2222.22
□строка□1
...
□строка□15
```

При составлении списка В/В следует учитывать ограничения:

- в списке В/В не может появляться перенимающий размер массив, но могут появляться его подобъекты (элементы и сечения);
- присутствующий в списке В/В размещаемый массив должен быть к моменту выполнения В/В размещен;
- все ссылки списка В/В к моменту выполнения В/В должны быть прикреплены к адресатам. Передача данных выполняется между файлом и адресатом;

- каждый конечный компонент присутствующего в списке В/В объекта производного типа не должен иметь атрибут PRIVATE;
- в списке В/В не могут присутствовать объекты производного типа, у которых среди компонентов какого-либо уровня есть ссылки.

Список В/В может быть пустым. Тогда при выводе создается запись нулевой длины. При вводе выполняется переход к следующей записи. Если же при пустом списке вывода используется формат, состоящий только из строки, то будет выведена запись, содержащая эту строку, например:

```
write(*, '(1x, "I am a test string")')
```

9.5.2. Циклические списки ввода/вывода

Список В/В может также содержать и циклический список, имеющий вид:

(*список объектов цикла*, *dovar* = *start*, *stop* [, *inc*])

где каждый *объект цикла* - это переменная (в случае ввода), или выражение (в случае вывода), или новый циклический список; *dovar* - переменная цикла - целая скалярная переменная; *start*, *stop*, *inc* - целые скалярные выражения. Циклический список оператора В/В работает так же, как и DO-цикл с параметром или неявный цикл оператора DATA и конструктора массива. Другое название циклического списка оператора В/В - *неявный цикл оператора В/В*.

Пример. Вывод горизонтальной линии.

```
print '(1x, 80a1)', ('_', k = 1, 80)  ! В списке вывода 80 элементов
```

9.5.3. Пример организации вывода

Задача: выполнить табуляцию функции двух переменных:

$$z = |x - y|\ e^{y/3}/(1/3 + \cos(x/y))$$

при изменении x от 1 до 5 с шагом 0.5, а y - от 1.1 до 1.5 с шагом 0.05.

Оформим результат в виде таблицы, содержащей заголовок, значения x по вертикали и значения y по горизонтали. В ячейках таблицы выведем соответствующие аргументам x и y значения z (рис. 9.1).

Зависимость z = ABS(x - y) * EXP(y/3)/(1/3 + cos(x/y))

$x \setminus y$	1.10	1.15	1.20	...	1.50
1.00	$z_{1,1}$	$z_{1,2}$	$z_{1,3}$	...	$z_{1,9}$
1.50	$z_{2,1}$	$z_{2,2}$	$z_{2,3}$	...	$z_{2,9}$
...					
5.00	$z_{11,1}$	$z_{11,2}$	$z_{11,3}$	...	$z_{11,9}$

Рис. 9.1. Проект таблицы вывода (выходная форма)

Ясно, что для организации такой таблицы потребуется выполнить некоторые преобразования: смещение позиции вывода, форматирование вывода значений *x*, *y* и *z*, вывод символьных данных.

Для вывода нам дополнительно надо знать:

- допустимое число выводимых на одной строке символов (в случае консоль-проекта это число равно 80);
- диапазон изменения значений функции *z*;
- необходимую точность представления *z* (число десятичных знаков).

Максимальное и минимальное значения *z*, а также точность представления *z* нужны для определения, во-первых, длины необходимого для вывода *z* поля и, во-вторых, способа представления *z* (в F- или E-форме). В общем случае эти данные могут быть определены лишь в процессе вычислений.

Рассмотрим подробно механизм формирования формата вывода одной строки таблицы. Положим, что максимальное и минимальное значения *z* могут быть размещены на поле длиной в 7 символов, причем два правых символа поля будут расположены после десятичной точки. Такое поле задается преобразованием F7.2. Расстояние между полями вывода *z* положим равным единице. Тогда при выводе одного поля следует использовать формат 1X, F7.2. Всего в одной строке таблицы будет размещено 9 полей со значениями *z*. Для их вывода необходим формат 9(1X, F7.2). Теперь предусмотрим при выводе строки значений *z* отступ от левой границы экрана в 1 символ и последующий вывод значения *x* в поле длиной в 5 символов, содержащее два десятичных знака. Получаем формат вывода строки таблицы: (2X, F5.2, 9(1X, F7.2)).

```
program zxy
 real :: x, y, xa = 1.0, xb = 5.0, ya = 1.1, yb = 1.51
 real :: z(10)                          ! Массив значений z для строки таблицы
 real :: dx = 0.5, dy = 0.05            ! Шаг изменения x и y
 character(80) :: title = 'Зависимость z = ABS(x - y) * EXP(y/3)/(1/3 + cos(x/y))'
 integer(1) k, tab
 tab = (80 - len_trim(title)) / 2
 write(*, '(<tab>x, a)') title          ! Вывод заголовка по центру экрана
 write(*, 1) ('_', k = 1, 80)           ! Вывод горизонтальной линии
 write(*, '(2x, a, 9f8.2)') 'x \ y', (y, y = ya, yb, dy)
 write(*, 1) ('_', k = 1, 80)           ! Вновь выводим горизонтальную линию
 x = xa
 do while(x <= xb)
  k = 0
  y = ya
  do while(y <= yb)                     ! Формирование массива значений z
   k = k + 1
   z(k) = abs(x - y) * exp(y / 3.0) / (1.0/3.0 + cos(x / y))
   y = y + dy
```

```
 end do                              ! Вывод строки таблицы
 write(*, '(2x, f5.2, 9(1x, f7.2))') x, z(:k)
 x = x + dx
end do
write(*, 1) ('_', k = 1, 80)
1 format(80a1)                       ! Формат вывода горизонтальной линии
end program zxy
```

Замечание. Для вывода значений *z* в одной строке в цикле по *y* можно использовать, применив дескриптор '\', непродвигающийся вывод. В этом случае можно обойтись без промежуточного массива *z*(1:10).

9.6. Согласование списка ввода/вывода и спецификации формата. Коэффициент повторения. Реверсия формата

Дескрипторы преобразований (ДП) подразделяются:

- на дескрипторы данных (ДД);
- на дескрипторы управления;
- на строки символов.

Дескрипторы данных, например F8.2 или I6, определяют размер и форму полей В/В, в которых размещаются текстовые представления данных. При форматном В/В каждому элементу списка В/В соответствует дескриптор данных. Элементы списка В/В и ДД должны быть согласованы по типам. Так, нельзя передать вещественное число, применяя преобразование I*w.m*. При вводе также должны быть согласованы внешние представления данных и ДД. Так, если поле ввода содержит символы и выполняется ввод с этого поля целого числа, то возникнет ошибка ввода.

Если в списке В/В присутствует несколько элементов, то каждый элемент выбирает один ДД из списка ДП. Правило выбора таково: *j*-й элемент списка В/В выбирает *j*-й ДД (назовем этот порядок выбора *правилом 1*). При этом поля всех элементов списка В/В располагаются в одной записи. Это правило работает, когда число ДД не меньше числа элементов в списке В/В.

Пример:

```
integer k, n, m(9)
read(*, '(I8, I5, I5, I5)') k, n, m(2), m(4)
```

Переменная *k* выберет дескриптор I8, остальные - I5. На входе должна быть определена запись с данными (символ □ использован для обозначения пробела):

```
□□□□□123□□345□□346□□347
```

Последовательность одинаковых ДД можно записать, использовав *коэффициент повторения* - задаваемую перед ДД целую буквальную константу без знака или задаваемое в угловых скобках целочисленное

выражение. Так, спецификацию формата в операторе ввода последнего примера можно записать компактнее:

```
read(*, '(I8, 3I5)') k, n, m(2), m(4)          ! 3 - коэффициент повторения
```

Коэффициент повторения может быть применен и для группы ДП. Общий вид записи повторяющейся группы ДП таков:

n[(] *группа* ДП [)]

Круглые скобки можно опустить, если *группа* ДП включает лишь один ДД. В группу ДП могут входить как ДД, так и дескрипторы управления. Использование коэффициента повторения перед дескриптором управления или строкой возможно лишь в том случае, когда дескриптор заключен в скобки. Использование коэффициента повторения отдельно перед дескрипторами управления и строками недопустимо.

Пример использования коэффициента повторения для группы ДП:

```
write(*, '(2x, F3.0, 2x, F3.0, 2x, F3.0)') a, b, c
write(*, '(3(2x, F3.0))') a, b, c
```

Теперь рассмотрим ситуацию, когда число ДД в спецификации формата меньше числа элементов в списке В/В. Пусть число ДД равно *m*. Тогда первые *m* элементов списка В/В выберут ДД по правилу 1. Далее начнется следующая запись (следующая строка текстового файла), и последующие *m* элементов списка В/В вновь выберут те же ДД, следуя правилу 1, и так далее до исчерпания списка В/В. Причем при вводе новая запись будет браться из файла, даже если введены не все данные последней передаваемой записи. Это, правда, верно, если в операторе В/В не задан спецификатор ADVANCE = 'NO', обеспечивающий передачу данных без продвижения. Назовем этот порядок выбора *правилом 2*.

Пример:

```
integer k, n, m(9)
read(*, '(I8, 3I5)') k, n, m(1:9)
```

В списке ввода 11 элементов. Переменная *k* выберет ДП I8; *n*, *m*(1), *m*(2) - I5; *m*(3) - I8; *m*(4), *m*(5), *m*(6) - I5; *m*(7) - I8, *m*(8), *m*(9) - I5. В файле данных должны быть определены не менее трех записей, например:

```
□□□□□123□□333□□444□□555□□-25
□□□□□777□□888□□999□□111□□222
□□□□□333□□444□□555
```

В результате ввода переменные получат значения: *k* - 123, *n* - 333, *m*(1) - - 444, *m*(2) - 555, *m*(3) - 777, *m*(4) - 888, *m*(5) - 999, *m*(6) - 111, *m*(7) - 333, *m*(8) - 444, *m*(9) - 555.

Правило 2 работает в том случае, когда один или несколько ДД не заключены в круглые скобки.

Круглые скобки употребляются, во-первых, если необходимо применить коэффициент повторения для последовательности ДД, во-вторых, чтобы установить формат В/В элементов списка В/В, для которых исчерпаны все ДД с учетом коэффициентов повторения.

Пример:

```
integer :: j, k, n, a(10), b(30)
read(*, '(2I8, 5(I2, I3), 5(I4, 1X, I1))') k, n, a, (b(j), j=1,30)
```

Число элементов в списке ввода равно 42. Число ДД с учетом коэффициентов повторения равно 22. Первые 22 элемента списка будут введены из первой записи файла, используя ДД по правилу 1. После ввода первых 22 элементов формат будет исчерпан. Для всех оставшихся записей будет применен последний заключенный в круглые скобки фрагмент формата - 5(I4, 1X, I1). Форматы 2I8 и 5(I2, I3) более использоваться не будут. Формат 5(I4, 1X, I1) будет применяться в соответствии с правилом 2, поэтому вторая и третья записи должны содержать не менее 10 полей данных каждая.

Общее правило использования формата при наличии в спецификации формата выделенных в скобки компонентов таково: если формат содержит заключенные в скобки ДД, то в случае, если он будет исчерпан, в файле возьмется новая запись и управление форматом вернется к левой скобке, соответствующей предпоследней правой скобке, или к соответствующему коэффициенту повторения, если он имеется. В приведенном примере - к 5(I4, 1X, I1). Это правило называется *реверсией формата*.

9.7. Дескрипторы данных

Рассмотрим теперь детально дескрипторы данных Фортрана. Полный перечень ДД приведен в табл. 9.2.

Таблица 9.2. Дескрипторы преобразования данных

Дескриптор	*Тип аргумента*	*Внешнее представление*
I*w*[.*m*]	Целый	Целое число
B*w*[.*m*]	"	Двоичное представление
O*w*[.*m*]	"	Восьмеричное представление
Z*w*[.*m*]	Любой	Шестнадцатеричное представление
F*w.d*	Вещественный	Вещественное число в F-форме
E*w.d*[E*e*]	"	" " в E-форме
EN*w.d*[E*e*]	"	" " "
D*w.d*	"	" " двойной точности
L*w*	Логический	T и F, .T и .F, .TRUE. и .FALSE.

A[*w*]	Символьный	Строка символов
G*w.d*[E*e*]	Любой	Зависит от типа данных

В таблице использованы следующие обозначения:

- *w* - длина поля, отведенного под представление элемента B/B;
- *m* - число ведущих нулей ($m \le w$);
- *d* - число цифр после десятичной точки ($d < w$).

Замечание. Фортран 95 позволяет задать значение *w*, равное нулю, например I0 или F0.5. В этом случае длина поля определяется значением выводимого числа. Это свойство применимо с дескрипторами B, F, I, O и Z. Если же $w > 0$ и при форматном выводе число полученных в результате преобразования символов превосходит *w*, то все поле заполняется звездочками (*). Например:

```
write(*, '(i0)') 123              ! 123
write(*, '(i2)') 123              ! **
write(*, '(f0.2)') 123.45         ! 123.45
write(*, '(f5.2)') 123.45         ! *****
end
```

Общие правила преобразования числовых данных:

- внешним представлением элемента B/B является строка символов;
- при вводе поле, полностью состоящее из пробелов, всегда интерпретируется как нуль. В противном случае интерпретация пробелов управляется дескрипторами BN и BZ;
- при вводе знак + может быть опущен;
- при вводе с дескрипторами F, E, G и D число цифр после запятой определяется положением десятичной точки. При ее отсутствии - значением параметра *d*;
- при выводе символы выравниваются по правой границе поля и при необходимости добавляются ведущими пробелами;
- если при выводе число полученных в результате преобразования символов превосходит длину поля *w*, то все поле заполняется звездочками;
- если вещественное число содержит больше цифр после десятичной точки, чем предусмотрено параметром *d*, то отображается округленное до *d* знаков после десятичной точки значение числа;
- при работе с комплексными числами необходимо применять одновременно два дескриптора вида F, E, G или D: первый - для действительной, второй - для мнимой части комплексного числа;
- дескрипторы управления и строки могут появляться между ДД;
- с дескрипторами F, E, G и D может быть использован дескриптор *k*P, где *k* - коэффициент масштабирования ($-127 \le k \le 127$). Действие

масштабного множителя *k*, если задан дескриптор *k*P, распространяется на все дескрипторы F, E, G и D списка до появления нового дескриптора *k*P;

- при чтении с дескрипторами I, B, O, Z, F, E, G, D или L входное поле может содержать запятую, которая завершает поле. При этом следующее поле начинается с символа, стоящего за запятой. Однако нельзя использовать в качестве разделителей запятые одновременно с дескрипторами позиционирования (T, TL, TR или *n*X), поскольку они изменяют позиции символов в записи.

Опишем теперь ДД.

При использовании дескриптора I*w*[.*m*] при *вводе* во внутреннее представление преобразовывается последовательность пробелов и цифр (со знаком или без знака), не содержащая десятичной точки или десятичной экспоненты. В списке *вывода* операторов WRITE и PRINT могут присутствовать элементы только целого типа. В противном случае возникнет ошибка выполнения.

Если задано положительное число *m*, то выводимое целое число будет дополнено *m* - *n* ведущими нулями, где *n* - число значащих цифр в числе. На ввод параметр *m* никакого влияния не оказывает.

```
integer :: k1 = 123, k2
read(*, '(I4)') k2                    ! Введем: -123
write(*, '(1X, I12, I12.7)') k1, k2   ! □□□□□□□□□123□□□□-0000123
```

B*w*[.*m*], O*w*[.*m*], Z*w*[.*m*] - двоичный (B), восьмеричный (O) и шестнадцатеричный (Z) дескрипторы данных. Данные, соответствующие этим дескрипторам, не могут содержать десятичной точки или знака (+ и -), но содержат пробелы или символы соответствующей системы счисления: цифры 0 и 1 при использовании дескриптора B; цифры 0-7 при использовании дескриптора O; цифры 0-9 и буквы *A* - *F* в случае дескриптора Z.

Дескрипторы B и O могут быть использованы только с целочисленными входными и выходными данными. Дескриптор Z может быть использован с данными любого типа. Кодировка чисел в B-, O- и Z-формах зависит от процессора (особенно отрицательных), поэтому программы, применяющие дескрипторы B, O и Z и соответствующие им формы данных, могут неадекватно работать на других компьютерах.

Параметр *w* задает длину поля B/B, а *m* - минимальное число выводимых символов ($m \le w$). При отсутствии *m* минимальное число выводимых символов равно единице. Если выход меньше, чем *w*, то он дополняется ведущими пробелами. Если выход меньше, чем *m*, то он дополняется ведущими нулями до размера *m*. Двоичные числа легче читать при наличии ведущих нулей вместо пробелов.

При вводе дескрипторы B, O и Z преобразовывают внешние двоичные, восьмеричные и шестнадцатеричные данные во внутреннее представление.

Каждый байт внутреннего представления соответствует восьми двоичным символам, трем восьмеричным и двум шестнадцатеричным. Например:

```
integer :: k(3) = 255
write(*, '(2x, b8, 1x, o3, 1x, z2)') k          ! 11111111  377  FF
```

Соответственно значение типа INTEGER(4) займет 32 двоичных, 12 восьмеричных и 8 шестнадцатеричных символов.

Если параметр *w* опущен, то длина поля B/B устанавливается по умолчанию: 8**n* - для формата B, 3**n* - для формата O и 2**n* - для формата Z, где *n* – значение параметра разновидности типа элемента B/B.

Порядок вывода символов элементов символьного типа совпадает с порядком их размещения в памяти. Байты числовых и логических типов выводятся в порядке их значимости слева направо (наиболее значимый байт выводится первым, т. е. расположен левее следующего по значимости байта).

Дескриптор Z может быть применен с символьными данными, если длина строки не превышает 130 символов. Если же передается строка большей длины, то будут преобразованы только первые 130 символов.

Вывод с применением дескрипторов B, O и Z выполняется по правилам (*n* - величина параметра разновидности типа):

- если *w* > 8**n* (B), 3**n* (O) или 2**n* (Z), то символы выравниваются по правой границе поля и добавляются ведущие пробелы, увеличивающие поле до *w* символов;
- если *w* ≤ 8**n* (B), 3**n* (O) или 2**n* (Z), то выводится *w* правых символов;
- если *m* > 8**n* (B), 3**n* (O) или 2**n* (Z), то символы выравниваются по правой границе поля и добавляются ведущие нули, увеличивающие число символов до *m*;
- если *m* < 8**n* (B), 3**n* (O) или 2**n* (Z), то параметр *m* не оказывает никакого действия.

Правила ввода (параметр *m* не оказывает никакого действия):

- если *w* ≥ 8**n* (B), 3**n* (O) или 2**n* (Z), то правые 8**n* (B), 3**n* (O) или 2**n* (Z) символов берутся из поля ввода;
- если *w* < 8**n* (B), 3**n* (O) или 2**n* (Z), то первые *w* символов читаются из поля ввода. Недостающие до длины 8**n* (B), 3**n* (O) или 2**n* (Z) символы замещаются пробелами.

В отличие от других ДД при выводе с дескрипторами B, O или Z значения, большего, чем можно разместить в поле вывода, выводятся не звездочки, а *w* правых символов. При вводе из незаполненных слева полей ввода знаковый бит игнорируется.

Пример:

```
character(2) :: st(3) = 'ab'
integer(2) :: k(3) = 3035
```

```
write(*, '(1x, z4.4, 1x, z2, 1x, z6)') st
write(*, '(1x, z4.4, 1x, z2, 1x, z6)') k
write(*, '(1x, b16.16, 1x, b2, 1x, b6)') k
write(*, '(1x, o5.5, 1x, o2, 1x, o6)') k
```

Результат:

```
6162   62     6162
0BDB  DB     BDB
0000101111011011 11 011011
05733 33 20005733
```

Расположенные в поле ввода завершающие пробелы трактуются как нули, если в операторе OPEN задан спецификатор BLANK = 'ZERO' или действует дескриптор BZ, например:

```
integer(1) :: k1, k2, k3
read(*, '(bn, b8, bz, b8, b8)') k1, k2, k3
write(*, '(1x, 3I5)') k1, k2, k3
```

Введем (символ □ используем для обозначения пробела):

```
□1□□□□□□□1□□□□□□□1000000
```

Результат:

```
  1   64   64
```

Дескриптор F*w.d* обеспечивает *вывод* вещественных чисел одиночной или двойной точности. Вывод выполняется на поле длиной в *w* символов. Один символ отводится под десятичную точку. При выводе отрицательного числа еще один символ будет отведен под знак. Из оставшихся *w* - 1 или *w* - 2 символов *d* символов будут отведены под числа, следующие после десятичной точки числа. Оставшиеся символы будут либо пробелами, либо цифрами, расположенными слева от десятичной точки. Выводимое число при преобразовании во внешнее представление при необходимости округляется.

При *вводе* с дескриптором F*w.d* передача данных осуществляется с поля длиной в *w* символов, на котором можно разместить целочисленные или вещественные числа в F- или E-форме (со знаком или без знака). Если десятичная точка отсутствует, то число десятичных знаков *вводимого* вещественного числа будет равно *d*. При наличии во внешнем представлении десятичной точки число десятичных знаков вводимой величины определяется положением десятичной точки и может отличаться от значения *d*. Пробелы между десятичными цифрами или между десятичной точкой и цифрами интерпретируются как нули, если задан дескриптор BZ, и игнорируются, если задан дескриптор BN или если оба дескриптора в спецификаторе формата отсутствуют.

Пример 1:

```
real a, b, c, d, e                ! Введем:
read(*, 1) a, b, c, d, e          ! □□□234□0.234□.234E2□-2.34E-3□.□□023
write(*, 1) a, b, c, d, e         ! □2.34□□□.23□□23.40□□□□-.0023□.00023
1 format(2F6.2, F7.2, F10.4, BZ, F7.5)
```

Пример 2. Все элементы массива *a* в результате ввода разных представлений числа 1.23 примут одно и то же значение.

```
real a(5)                         ! Вводимые данные:
read(*, 1) a                      ! □□12300□1.23□□□□12.3E-1□1□□2300□.0123E2
write(*, 1) a                     ! 1.2300  1.2300  1.2300  1.2300  1.2300
1 format(10F8.4)
```

Замечание. В последнем примере лучше воспользоваться выводом, управляемым списком В/В, указав во входном потоке, например, так:

```
read(*, *) a                      ! 1.23 1.23 1.23 1.23 1.23
```

или так (поля данных разделяются запятой):

```
read(*, *) a                      ! 1.23, 1.23, 1.23, 1.23, 1.23
```

или так:

```
read(*, *) a                      ! 5*1.23
```

Рассмотрим механизм преобразования числа -1.23 при выводе на примере дескриптора F8.3, т. е. результат выполнения оператора

```
write(*,'(f8.3)') -1.23           ! □-1.230
```

Число -1.23 расположится на поле длиной в 8 символов. Поскольку $d = 3$, а в числе только две цифры после десятичной точки, то последним символом будет 0, далее последуют символы 3, 2, десятичная точка, 1 и знак -. Первыми двумя символами в отведенном под число поле будут пробелы. Правда, первый пробел на экране не отобразится и поэтому будет выведено □-1.230.

Применение дескриптора масштабирования *k*P оказывает следующие действия:

- при *вводе* дескрипторы *k*PF*w.d* означают, что после преобразования F*w.d* введенное число будет умножено на 10^{-k};
- при *выводе* с форматом *k*PF*w.d* выводимая величина прежде умножается на 10^{k}, а затем выводится в соответствии с преобразованием F*w.d*.

Пример:

```
write(*, '(5PF13.4)') -1.23       ! -123000.0000
write(*, '(F13.4)') -1.23         ! □□□□□-1.2300
```

Если выводимое значение не может быть размещено в отведенном поле, то результатом вывода будут звездочки.

Дескриптор преобразования вещественных чисел E*w.d*[Ee] требует, чтобы при *выводе* ассоциируемый с дескриптором E элемент имел вещественный тип одиночной или двойной точности.

При *вводе* входное поле идентично входному полю дескриптора F. Параметр *e* дескриптора E при вводе игнорируется.

Форма выходного поля зависит от задаваемого дескриптором *k*P коэффициента масштабирования. При равном нулю коэффициенте масштабирования (задается по умолчанию) выходное поле, длина которого равна *w*, представляет собой: знак минус (в случае отрицательного числа), далее десятичная точка, затем строка из *d* цифр, затем поле под десятичную экспоненту, имеющее одну из показанных в табл. 9.3 форм.

Таблица 9.3. Форма поля под десятичную экспоненту в дескрипторе E

Дескриптор	*Показатель степени экспоненты*	*Форма поля*
E*w.d*	$\|p\| \le 99$	E, затем плюс или минус, затем показатель степени десятичной экспоненты из двух цифр
E*w.d*	$99 < \|p\| \le 999$	Плюс или минус, затем показатель степени десятичной экспоненты из трех цифр
E*w.d*E*e*	$\|p\| \le 10^e - 1$	E, затем плюс или минус, затем показатель степени десятичной экспоненты из *e* цифр, который может содержать и ведущие нули

Пример:

```
real(8) :: a = 1.23D+205
real(4) :: b = -.0000123445, c = -.123445
write(*, '(E15.8)') a              !  .12300000+206
write(*, '(1x,2E12.5)') b, c       ! -.12344E-04 -.12344E+00
write(*, '(1x,2E14.5E4)') b, c     ! -.12344E-0004 -.12344E+0000
```

Рассмотрим механизм преобразования числа 1.23 при выводе на примере дескриптора E11.5, т. е. результат выполнения оператора

```
write(*, '(e11.5)') 1.23           ! □.12300E+01
```

Число 1.23 расположится на поле длиной в 11 символов. Последние 4 символа в дескрипторе E отводятся для обозначения десятичной экспоненты (E), знака и показателя степени. При выводе все цифры отображаются после десятичной точки, т. е. на выходе мы получим число $0.123*10^1$. Поскольку в дескрипторе после десятичной точки предусмотрено 5 символов (*d* = 5), то после вывода .123 будут добавлены два нуля, а затем уже последует десятичная экспонента E+01. Результатом преобразований будет строка □.12300E+01.

Дескриптор *k*P с дескриптором E*w.d*[Ee] работает так:

- если масштабный коэффициент *k* больше -*d* и $k \le 0$ ($-d < k \le 0$), то выходное поле содержит *k* ведущих нулей после десятичной точки и *d*+*k* значащих цифр после них;
- если $0 < k < d + 2$, то выходное поле содержит *k* значащих цифр слева от десятичной точки и *d* - *k* - 1 цифр будут расположены после десятичной точки. Другие значения *k* недопустимы, например:

```
real :: b = -.0000123445, c = -.123445
write(*, '(1x,  2PE12.5)') b        !  -12.3445E-06
write(*, '(1x, -2PE12.5)') c        ! □-.00123E+02
```

Дескриптор EN*w.d*[E*e*] передает данные в *инженерном* формате и работает так же, как и дескриптор E, за тем исключением, что при *выводе* абсолютное значение неэкспоненциальной части всегда находится в диапазоне от 1 до 1000. Показатель степени экспоненты при работе с дескриптором EN всегда кратен трем. Форма поля под экспоненту в дескрипторе EN такая же, как и для дескриптора E, например:

```
real :: x = -12345.678, y = 0.456789, z = 7.89123e+23
write(*, 1) x, z                    !-12.34568E+03 789.12300E+21
1 format (1x, en13.5, 1x, en13.5)
write(*, 2) y, z                    ! 456.79e-0003  789.12e+0021
2 format (1x, en13.2e4, 1x, en13.2e4)
```

Дескриптор ES*w.d*[E*e*] обеспечивает передачу данных в *научном* формате и работает так же, как и дескриптор E, за тем исключением, что при *выводе* абсолютное значение неэкспоненциальной части всегда находится в диапазоне от 1 до 10. Форма поля под экспоненту при работе с ES такая же, как и для дескриптора E.

```
real :: x = -12345.678,  y = 0.456789, z = 7.89123e+23
write(*, 1) x, z                    ! -1.23457E+04   7.89123E+23
1 format (1x, es13.5, 1x, es13.5)
write(*, 2) y, z                    !  4.57E-0001    7.89E+0023
2 format (1x, es13.2e4, 1x, es13.2e4)
```

Элементы списка *вывода*, ассоциируемые с дескриптором преобразования двойной точности D*w.d*, должны иметь вещественный тип одиночной или двойной точности. Все правила и параметры, применимые к дескриптору E, также применимы и к дескриптору D.

Входные поля при работе с дескриптором D формируются так же, как и входные поля для дескриптора F, с теми же значениями параметров *w* и *d*.

Форма выходного поля зависит от масштабного коэффициента, задаваемого дескриптором *k*P. При равном нулю коэффициенте масштабирования выходное поле выглядит так: знак минус (в случае вывода отрицательного числа), затем десятичная точка, затем строка цифр

и, наконец, поле под десятичную экспоненту. Последнее поле формируется по одному из указанных в табл. 9.4 правил.

Таблица 9.4. Форма поля под десятичную экспоненту в дескрипторе D

Дескриптор	*Показатель степени экспоненты*	*Форма поля*
D*w.d*	$\|p\| \le 99$	D, затем плюс или минус, затем показатель степени десятичной экспоненты из двух цифр
D*w.d*	$99 < \|p\| \le 999$	Плюс или минус, затем показатель степени десятичной экспоненты из трех цифр

Масштабирование при работе с дескриптором D выполняется по тем же правилам, по которым оно выполняется и для дескриптора E.

Пример:

```
real(8) :: b = -.0000123445_8
write(*, '(1x, D12.5)') b              !  -.12344D-04
write(*, '(1x, 2PD12.5)') b            !  -12.3445D-06
write(*, '(1x,-2PD12.5)') b            !  -.00123D-02
```

При передаче данных логического типа используется дескриптор L*w*. Если ассоциируемый в списке *вывода* с дескриптором L элемент не является элементом логического типа, то возникнет ошибка исполнения. В результате преобразования значения логического типа будет выведено: *w* - 1 пробелов, а затем T или F.

Поле *ввода*, так же как и поле вывода, имеет длину в *w* символов и может содержать пробелы, затем необязательную десятичную точку, затем T (t) для задания *истина* или F (f) для задания *ложь*. Любые последующие символы в поле ввода игнорируются. Поэтому на входе может быть задано и .TRUE. и .FALSE.

Пример:

```
logical :: fl = .true., yesno = .false.
write(*, '(1X, 2L5)') fl, yesno        !  T    F
```

Дескриптор A[*w*] используется преимущественно при В/В данных символьного типа. Если длина *w* опущена, то она принимается равной длине ассоциируемого с дескриптором A элемента В/В.

Элемент списка В/В может быть любого типа. Если он не является элементом символьного типа, то каждому байту внутреннего представления ставится в соответствие символ. Например, элементу типа INTEGER(2) соответствует 2 символа. Однако независимо от используемого типа данных каждый элемент списка В/В должен быть задан как последовательность символов.

Когда элемент списка В/В имеет тип INTEGER, REAL или LOGICAL, то для задания строк символов можно использовать холлеритовские символьные константы. Для каждого типа данных сохраняется возможность использования встроенных для данного типа операций. Так, объявленные в INTEGER строки символов можно складывать, перемножать и т. д.

Входная строка текста вводится посимвольно с последующим преобразованием символа в его двоичное представление.

Если при *вводе* число символов элемента $k < w$, то будет введено w символов, но только k последних будут принадлежать элементу ввода. Если $k > w$, то будет введено w символов, а оставшиеся символы строки будут заполнены завершающими пробелами.

Выводимые с применением дескриптора A данные выравниваются по правой границе поля, завершающие пробелы сохраняются.

Пример:

```
integer(4) :: b = '2bcd7', g = 4Ha25f
real(8) :: d = '#456&7xz'
character(12) :: st1 = 'string 1', st2 = 10hNew string, st3*6
write(*, '(4(1x, A))') b, g + 2, d          !2bcd c25f #456&7xz
write(*, '(4(1x, A))') b - g                ! -/._
write(*,'(1x, a14, a5)') st1, st2           !  string 1    New s
read(*, '(A3)') st3                         !ert - строка ввода (k > w)
write(*, *) st3, 's'                        !ert   s
read(*, '(A8)') st3                         !ert - строка ввода (k < w)
write(*, *) st3, 's'                        !t     s
```

Пояснение. Переменная *b* занимает в памяти ЭВМ 4 байта, поэтому из строки '2bcd7' в *b* будет установлено только 4 первых ее символа.

Обобщающий дескриптор G*w.d*[E*e*] может быть использован с данными любого встроенного типа. Для целочисленных данных дескриптор G*w.d* имеет такое же действие, как и дескриптор I*w.m*. Для логических данных G*w.d* действует так же, как и L*w*. Для символьных данных G*w.d* действует так же, как и A*w*.

Пример:

```
integer(4) :: k = 355
logical :: fl = .true.
character(10) :: st = '  string'
write(*, '(1x, 3g10.5)') k, fl, st          !  00355        T  string
```

Для вещественных данных дескриптор G*w.d*[E*e*] более гибок, чем дескриптор F, поскольку автоматически переключается с формата F на формат E в зависимости от величины передаваемых данных.

Когда G*w.d*[E*e*] используется как вещественный дескриптор, поле ввода равно w символам и d символов на этом поле отводятся под следующие за десятичной точкой числа, т. е. при *вводе* G*w.d*[E*e*] работает так же, как

и F*w.d*. При *выводе* преобразование G в зависимости от значения выводимой величины соответствует либо F-, либо E-преобразованию. В табл. 9.5, 9.6 приведена интерпретация дескрипторов G и GE при выводе.

*Таблица 9.5. Интерпретация дескриптора G*w.d *при выводе*

Абсолютное значение величины	*Интерпретация*
$x < 0.1$	Gw.d = Ew.d
$0.1 \le x < 1$	Gw.d = F(w - 4).d, 4(□)
$1 \le x < 10$	Gw.d = F(w - 4).(d - 1), 4(□)
$10^{d-2} \le x < 10^{d-1}$	Gw.d = F(w - 4).1, 4(□)
$10^{d-1} \le x < 10^{d}$	Gw.d = F(w - 4).0, 4(□)
$10^{d} \le x$	Gw.d = Ew.d

Дескриптор G*w.d*[D*e*] эквивалентен дескриптору G*w.d*[E*e*] за тем исключением, что при выводе вместо E печатается D.

*Таблица 9.6. Интерпретация дескриптора G*w.d*E*e *при выводе*

Абсолютное значение величины	*Интерпретация*
$x < 0.1$	Gw.dEe = Ew.d
$0.1 \le x < 1$	Gw.dEe = F(w - e - 2).d, (e + 2)(□)
$1 \le x < 10$	Gw.dEe = F(w-e-2).(d-1),(e+2)(□)
$10^{d-2} \le x < 10^{d-1}$	Gw.dEe = F(w - e - 2).1, (e + 2)(□)
$10^{d-1} \le x < 10^{d}$	Gw.dEe = F(w - e - 2).0, (e + 2)(□)
$10^{d} \le x$	Gw.dEe = Ew.d

Пример:

```
real :: b1 = .01234, b2 = 123400, b3 = 123.4
write(*, 1) b1, b2, b3          !  -.12340E-01  .12340E+06  123.40
write(*, 2) b1, b2, b3          ! -.12340E-001 .12340E+006 123.40
1 format(1x, 3G12.5)
2 format(1x, 3G12.5E3)
```

9.8. Дескрипторы управления

Дескрипторы управления также называют *неповторяющимися дескрипторами преобразований* (ДП), поскольку перед такими дескрипторами (если только дескриптор не заключен в скобки) в спецификации формата нельзя указать коэффициент повторения.

Неповторяющиеся ДП служат:

- для управления позицией В/В (преобразования *n*X, T, TL, TR);

- внесения в запись дополнительной информации (преобразования апострофа и Холлерита);
- масштабирования данных и других, приведенных в табл. 9.7, функций управления В/В.

В списке ДП для разделения его отдельных дескрипторов используется запятая, которая может быть опущена:

- между дескриптором Р и сразу следующими за ним дескрипторами F, E, EN, ES, D или G, например: 1X, 2P F9.6;
- перед или после дескрипторов апострофа ('), кавычек ("), обратного слеша (\) или двоеточия (:), например: 1x, I3, ' ' B8 \;
- перед и после слеша, например: 1x, 2I5, 2(/ 2F5.2).

Таблица 9.7. Неповторяющиеся дескрипторы преобразований

Формы	*Имя*	*Назначение*	*Использование*
Строка	Преобразование апострофа	Передает строку текста в файл	Вывод
nH	Преобразование Холлерита	Передает *n* символов в файл	"
Q	Преобразование опроса	Возвращает число непрочитанных символов записи	Ввод
Tn, TLn, TRn	Преобразование позиции	Спецификация позиции в записи	В/В
nX	Преобразование позиции	" " " "	"
SP, SS, S	Преобразование знака плюс	Управление выводом знака плюс	Вывод
/	Преобразование слеша	Переход к следующей записи и простановка символов конца записи	В/В
\	Преобразование обратного слеша	Продолжение текущей записи (для тех же целей можно использовать знак $)	Вывод
:	Прерывание выполнения действия ДП	При исчерпании списка вывода прерывает выполнение ДП	"
kP	Преобразование масштабного коэффициента	Устанавливает значение показателя степени в ДД F, E, D и G	В/В
BN, BZ	Интерпретация пробела	Устанавливает способ интерпретации пробелов	Ввод

Преобразование апострофа или *двойных кавычек* выполняет вывод заключенной в апострофы или кавычки строки. Для вывода обрамленной апострофами и содержащей апострофы строки необходимо указать каждый выводимый апостроф дважды (либо заключить строку в двойные кавычки). Аналогично выполняется вывод содержащих двойные кавычки строк. Преобразование *апострофа* и *двойных кавычек* не может быть использовано с оператором READ.

Пример:

```
write(*, 1)
1 format(2x, 'Введите границы отрезка [a, b]: ')
! или
write(*, '(2x, "Введите границы отрезка [a, b]: ")')
! или, применив дескриптор A
write(*, '(2x, a)') 'Введите границы отрезка [a, b]: '
read(*, *) a, b
```

Замечание. Если в спецификации формата оператора WRITE выводимая строка заключена в апострофы, то сама спецификация должна быть заключена в двойные кавычки; можно сделать наоборот, например:

```
write(*, "(2x, 'Введите границы отрезка [a, b]: ')")
```

Преобразование Холлерита. Дескриптор *n*H передает *n* символов, включая пробелы, в файл или на экран. Число символов, следующих за дескриптором *n*H, должно быть равно *n*. Преобразование Холлерита может быть использовано везде, где допустимо применение символьных констант. Принято называть константы, определенные при помощи дескриптора *n*H, *холлеритовскими константами*.

Пример:

```
write(*, 1)
1 format(2x, 31HВведите границы отрезка [a, b]:)
! или
write(*, '(2x, 31HВведите границы отрезка [a, b]:)')
```

Замечание. Фортран 95 удалил холлеритовские константы из стандарта; несмотря на это, они продолжают поддерживаться CVF.

Преобразование опроса. Дескриптор Q возвращает число непереданных символов записи. Соответствующий дескриптору Q элемент списка В/В должен быть целого или логического типа.

В следующем примере дескриптору Q соответствует переменная *nq*, в которую благодаря дескриптору Q будет считано (после ввода пяти элементов массива *kar*) число непереданных символов записи. Затем значение *nq* будет использовано при вводе массива *chr*.

```
integer kar(5), nq
character(1) chr(80)
read(4, '(5I4, Q, 80A1)') kar, nq, (chr(i), i= 1, min(nq, 80))
```

Возвращаемое дескриптором Q значение можно использовать не только в текущем, но и в следующем операторе ввода. Для этого после опроса записи надо остаться в текущей записи, т. е. применить ввод без продвижения, например:

```
integer k, nq                    ! Спецификатор ADVANCE = 'NO' задает ввод
character(1) chr(80)             ! без продвижения
read(*, '(I2, Q)', advance = 'no') k, nq
read(*, '(80A1)') (chr(i), i= 1, min(nq, 80))
```

Преобразование позиции. Дескрипторы T, TL и TR задают позицию записи, в которую или из которой будет передаваться следующий символ. Новая позиция может быть задана как левее, так и правее текущей. Это позволяет при вводе использовать запись более одного раза. Правда, не рекомендуется перемещаться в обратном направлении более чем на 512 байт (символов).

Дескриптор T*n* задает абсолютную табуляцию: передача следующего символа будет выполняться начиная с позиции *n* (отсчет позиций выполняется от начала записи).

Дескриптор TR*n* задает относительную правую табуляцию: передача следующего символа будет выполняться начиная с позиции, расположенной на *n* символов правее текущей позиции.

Дескриптор TL*n* задает относительную левую табуляцию: передача следующего символа будет выполняться начиная с позиции, расположенной на *n* символов левее текущей позиции. Если задаваемая дескриптором TL*n* позиция оказывается перед первой позицией текущей записи, то передача следующего символа будет выполняться с первой позиции. Если размер записи больше выделенного для В/В буфера, то нельзя выполнить левую табуляцию в позицию, принадлежащую предыдущему буферу.

Если в результате применения дескриптора позиционирования выполнено перемещение правее последнего переданного символа и выполнен вывод нового значения, то пространство между концом предыдущего значения и началом нового значения будет заполнено пробелами.

Дескриптор *n*X используется для перемещения позиции В/В на *n* символов вперед.

Пример:

```
real :: a = 1.23, b = 5.78, c
write(*, 1) a, b                 ! □□□□□□a =  1.230□□□□□□□b =  5.780
1 format(T7, 'a = ', f6.3, TR7, 'b = ', f6.3)
```

```
read(*, '(20(f6.2, TL6))') a, b, c   ! Введем: □□4.67
write(*, '(4x, 3f6.2)') a, b, c      ! □□□□□4.67□□4.67□□4.67
```

Управление выводом знака + в числовых полях выполняется при помощи дескрипторов SP, SS и S. Использование дескриптора SP обеспечивает вывод знака + в числовых полях, в которых выводятся положительные числа. Дескриптор SS подавляет вывод знака + (принимается по умолчанию). Дескриптор S восстанавливает действие дескриптора SS.

Пример:

```
real :: a = 1.23, b = 5.78
write(*, '(2f6.2)') a, b              ! □1.23□□5.78
write(*, '(sp, 2f6.2)') a, b          !+1.23□+5.78
write(*, '(sp, f6.2, s, f6.2))') a, b  !+1.23□□5.78
```

Преобразование слеша. В текущей записи *слеш* (/) указывает на конец подлежащих передаче данных.

При *вводе* слеш позиционирует файл за текущей записью.

При *выводе* слеш обеспечивает простановку символов конца записи и позиционирует файл за этими символами. Перед слешем может быть задан коэффициент повторения.

Пример:

```
integer a(20)
open(9, file = 'a.txt', blank = 'null')
read(9, 1) a
write(*, 1) a
1 format(7i3 / 5i3 / 8i3)
```

Состав файла a.txt:

```
  1   2   3   4   5   6   7   8   9  10
 11  12  13  14  15  16  17  18  19  20
 -1  -2  -3  -4  -5  -6  -7  -8  -9 -10
```

При вводе по формату 1 из первой записи файла будут введены 7 чисел, из второй - 5 и из третьей - 8. Переход с одной записи на следующую обеспечивается преобразованием слеша.

Результат вывода по формату 1:

```
  1   2   3   4   5   6   7
 11  12  13  14  15
 -1  -2  -3  -4  -5  -6  -7  -8
```

Преобразование обратного слеша. По умолчанию при завершении передачи данных файл позиционируется вслед за обработанной записью, даже если переданы не все данные записи. Однако если последовательность ДП содержит *обратный слеш* (\), то продвижения файлового указателя при завершении исполнения оператора вывода не произойдет. Поэтому

следующий оператор вывода продолжит передачу данных в ту же запись, начиная с той позиции, в которой файл был оставлен последним оператором вывода.

Такое же действие оказывает и дескриптор $, а также спецификатор оператора B/B ADVANCE = 'NO', который, правда, в отличие от дескрипторов \ и $ применима и при вводе. Дескрипторы $ и \ часто применяются для организации запросов, например:

```
write(*, 1)
read(*, *) a, b
1 format( 2x, 'Введите границы отрезка [a, b]: ', \)
! или format( 2x, 'Введите границы отрезка [a, b]: ', $)
```

После вывода запроса по формату 1 ввод границ отрезка можно будет выполнить на той же строке, где выведен и запрос.

Прерывание выполнения действия ДП спецификации формата осуществляется дескриптором *двоеточие* (:). Прерывание происходит в том случае, когда список *вывода* исчерпан.

Пример, когда двоеточие прерывает вывод пояснительного текста:

```
real :: a = 0.59, eps = 1e -5
write(*, 1) a, eps                ! Начало отрезка .59 Точность: .100E-04
write(*, 1) a                     ! Начало отрезка .59
1 format(1x, 'Начало отрезка: ', f4.2, : 2x, 'Точность: ', e9.3)
```

Преобразование масштабного коэффициента задается дескриптором *k*P. Дескриптор устанавливает коэффициент масштабирования для всей последовательности ДД F, E, D и G до тех пор, пока не встречен другой дескриптор *k*P. По умолчанию преобразование масштабного коэффициента не задано. Действие дескриптора *k*P было описано при рассмотрении ДД F и E. Здесь же мы ограничимся примером:

```
real a(4), b(4), c(4), d(4)
open(9, file = 'a.txt')
read(9, 1) (a(i), b(i), c(i), d(i), i = 1, 4)
1 format(f10.6, 1p, f10.6, f10.6, -2p, f10.6)
write(*, 2) (a(i), b(i), c(i), d(i), i = 1, 4)
2 format (4f11.3)
```

Состав файла a.txt:

```
12340000  12340000  12340000  12340000
   12.34     12.34     12.34     12.34
 12.34e0   12.34e0   12.34e0   12.34e0
 12.34e3   12.34e3   12.34e3   12.34e3
```

Результат:

```
  12.340     1.234     1.234  1234.000
  12.340     1.234     1.234  1234.000
```

```
  12.340     12.340     12.340     12.340
12340.000 12340.000 12340.000 12340.000
```

Интерпретация пробела в числовых полях управляется ДП BN и BZ.

Дескриптор BN игнорирует пробелы: в поле выбираются все отличные от пробелов символы и выравниваются по правой границе. Так, в случае применения BN поля -1□□□.23 и □□□□-1.23 эквивалентны.

Дескриптор BZ идентифицирует все пробелы поля как нули. Так, в случае BZ поля 1□□□□.2□□3 и 100.203 эквивалентны.

Замечание. Пробелы, следующие после E или D, при вводе вещественных чисел всегда игнорируются, независимо от вида примененного дескриптора интерпретации пробелов.

По умолчанию первоначально операторы B/B интерпретируют пробелы в соответствии с дескриптором BN, если только в операторе OPEN не задан спецификатор BLANK = 'NULL' | 'ZERO'.

Если задан дескриптор BZ, то он будет действовать до тех пор, пока не будет обнаружен дескриптор BN.

9.9. Управляемый списком ввод/вывод

При управляемом списком B/B все преобразования выполняются с учетом типа элементов списка B/B и значений передаваемых данных в соответствии с принятыми в Фортране соглашениями.

Управляемый списком B/B применяется при работе с текстовыми последовательными устройствами и не может быть использован при работе с неформатными файлами и с форматными файлами, подсоединенными для прямого доступа.

Различают два вида управляемого списком B/B: управляемый именованным и неименованным списком.

При использовании неименованного списка передача данных может выполняться и во внутренние файлы.

9.9.1. Управляемый именованным списком ввод/вывод

Синтаксис B/B под управлением именованного списка:

WRITE(*u*, [NML =] *имя списка* B/B)

READ(*u*, [NML =] *имя списка* B/B)

u - устройство B/B (см. разд. 10.2).

имя списка B/B - задается оператором NAMELIST.

Спецификатор NML может быть опущен. Его присутствие обязательно, если заданы другие спецификаторы оператора WRITE или READ, например END.

Управляемый списком B/B удобен на этапах отладки и тестирования программы, когда часто нужно вывести имена переменных и их значения.

9.9.1.1. Объявление именованного списка

Оператор NAMELIST объявления именованного списка В/В должен появляться в разделе объявлений программной единицы и имеет вид:

NAMELIST / *имя списка* В/В / *список переменных* &
[/ *имя списка* В/В / *список переменных* ...]

имя списка В/В - имя списка переменных. Одно и то же *имя списка* В/В может появляться в операторе NAMELIST неоднократно. В этом случае соответствующие именам списки переменных рассматриваются как один список. Порядок расположения элементов в таком списке соответствует их расположению в операторе NAMELIST.

список переменных - список имен переменных, может содержать переменные производного типа, которые, правда, не должны в качестве компонентов иметь ссылки. Формальные параметры не могут быть элементами списка. Также элементами списка не могут быть подобъекты (сечения массивов, подстроки...). Одно и то же имя может появляться более чем в одном *списке переменных.*

Оператор NAMELIST присваивает имя списку переменных. Далее это имя используется в операторах В/В. Например:

```
integer :: ia = 1, ib = 2
complex :: z(2) = (/ (2.0, -2.0), (3.5, - 3.5) /)
namelist /ico/ ia, ib, z          ! Объявляем именованный список
write(*, ico)                     ! Выводим именованный список на экран
```

9.9.1.2. NAMELIST-вывод

При выводе именованного списка результат имеет вид:

```
&имя_списка_вывода
имя переменной = значение | список значений
...
имя переменной = значение | список значений
/
```

Пример:

```
integer :: k, iar(5) = (/ 41, 42, 43, 44, 45 /)
logical :: fl = .true.
real :: r4*4 = 24, r8*8 = 28
complex(4) :: z4 = (38.0, 0.0)
character(10) :: c10 = 'abcdefgh'
type pair
 character(1) a, b
end type pair
type(pair) :: cp = pair('A', 'B')
namelist /mesh/ k, fl, r4, r8, z4, c10, cp, iar
k = 100
write(*, mesh)
```

Результат:

```
&MESH
K =         100
FL =  T
R4 =       24.000000
R8 =       28.000000000000000
Z4 =         (38.000000,0.000000E+00)
C10 =  abcdefgh
CP =   A B
IAR =          41 42 43 44 45
/
```

Из примера видно, что символьные данные выведены без обрамляющих кавычек. При необходимости можно вывести строку с обрамляющими кавычками или апострофами. Для этого текстовой файл (или последовательное устройство, например экран) надо открыть с DELIM = 'APOSTROPHE' или DELIM = 'QUOTE', которые задают вид ограничителя символьных данных: апостроф (') или кавычки ("). При NAMELIST-выводе в файл, открытый, например, с DELIM = 'APOSTROPHE', выводимые символьные данные ограничиваются апострофом, а присутствующие в строке апострофы удваиваются. Аналогичное влияние оказывает спецификатор DELIM = 'QUOTE'. Отсутствие в операторе OPEN спецификатора DELIM эквивалентно заданию в нем спецификатора DELIM = 'NONE'.

Пример:

```
integer :: k = 100
character(10) :: c10 = 'abcd"efgh'
type pair
 character(1) a, b
end type pair
type(pair) :: cp = pair('A', 'B')
namelist /mesh2/ k, c10, cp
open(10, file = 'a.txt', delim = 'quote')
write(10, mesh2)                    ! Вывод в файл a.txt
open(6, delim = 'quote')            ! Изменяем свойства подсоединения
write(6, mesh2)                     ! Вывод на экран
```

Результат (кавычки внутри строки C10 удваиваются):

```
&MESH2
 K =         100
 C10 =  "abcd""efgh"
 CP =   "A""B"
/
```

9.9.1.3. NAMELIST-ввод

Ввод именованного списка практически зеркально противоположен его выводу.

При вводе именованного списка оператор ввода ищет в файле начало списка, которое может иметь вид: *&имя_списка* или $*имя_списка*. Перечень принадлежащих именованному списку данных завершается слешем (/) или знаком доллара ($) или амперсанда (&). После знаков доллара и амперсанда может следовать слово END. Каждый элемент ввода имеет вид:

имя_переменной = значение | список значений

имя_переменной (хотя в NAMELIST могут присутствовать только полные объекты) может при вводе быть и подобъектом - сечением или элементом массива, подстрокой, компонентом записи...

Пример:

```
&eli k = 1 /
$eli k = 1 $        или    $eli k = 1 $end
&eli k = 1 &        или    &eli k = 1 &end
$eli k = 1 &end     или    &eli k = 1 $end
```

Порядок, в котором появляются имена переменных в файле, не имеет значения. Количество перечисленных входных данных может быть меньше заявленного. Имена переменных и массивов в файле должны совпадать с соответствующими именами *списка переменных* оператора NAMELIST. Разделителями между входными данными являются запятая, пробел, символ конца строки и знак табуляции. Это значит, что в одной строке файла может располагаться более одного элемента ввода.

Пример:

```
integer :: k, iar(5)
logical :: fl
real r4
complex z4
character :: c10*10, c4*4
namelist /mesh/ k, fl, r4, z4, c10, c4, iar
open(1, file = 'a.txt')
read(1, mesh)
write(*, *) k, iar, fl
write(*, *) r4, z4, ' ', c10, ' ', c4
```

Состав файла a.txt:

```
&Mesh   K = 100,  FL =  T, Z4 = (38, 0),  C10 = 'abcdefgh'
        r4 = 24.0, iar = 1, 2, 3, 5, 5, c4 = 'sub'
/
```

Результат:

```
100          1       2              3              5          5     T
   24.000000       (38.000000,0.000000E+00)          abcdefgh      sub
```

Замечание. Именованный, предназначенный для ввода список с данными теперь, согласно стандарту 1995 г., может содержать комментарий, следующий, как и в исходном коде, после восклицательного знака, например:

Состав файла a.txt:

```
&Mesh   K = 100, FL =  T, Z4 = (38, 0),      ! Задание числовых данных
        C10 = 'abcdefgh'                     ! Значение символьной переменной
        r4 = 24.0, iar = 1, 2, 3, 5, 5 /
```

Если в *списке значений* (в примере такой список использован для задания значений массива *iar*) перед первой запятой или между запятыми отсутствует значение, то оно трактуется как *null* и значение соответствующего элемента списка ввода не изменяется.

Для задания логической величины в файле следует указать T или .TRUE., F или .FALSE. или иное удовлетворяющее дескриптору L значение.

Символьные данные могут быть заданы без ограничителя, однако если строка содержит пробелы, или запятые, или слеш, или символы конца строки, то для правильного ввода ее необходимо заключить в кавычки или апострофы. При этом присутствующие в строке ограничители должны быть удвоены.

Пример:

```
integer :: iar(5) = 100
logical fl
character(10) st
namelist /mesh2/ iar, fl, st
open(1, file = 'a.txt')
read(1, mesh2)
write(*, *) iar, fl, ' ', st
```

Состав файла a.txt (в файле заданы подобъекты массива *iar*, причем элементам *iar*(1) и *iar*(3) соответствуют значения):

```
&Mesh2  st = 'ab d''ef gh'
  iar(1:4) = , -2, , -4, iar(5) = 55, fl = .False.
&end
```

Результат:

```
100        -2        100        -4        55 F  ab  d'ef  gh
```

Замечание. Повторяющиеся значения *списка значений* можно записать в виде одного значения, проставив перед ним коэффициент повторения, после которого следует звездочка (*). Например, задание:

iar = 3*5, 2*10

аналогично следующему:

iar = 5, 5, 5, 10, 10

9.9.2. Управляемый неименованным списком ввод/вывод

В случае неименованного списка операторы В/В имеют вид:

WRITE(*u*, [FMT =] *) [*список вывода*]

PRINT * [, *список вывода*]

READ(*u*, [FMT =] *) [*список ввода*]

READ * [, *список ввода*]

u - устройство В/В (см. разд. 10.2 и 10.3);

* - указывает на то, что В/В будет управляться списком В/В.

Список В/В формируется по тем же правилам, которые действуют и при форматном В/В.

9.9.2.1. Управляемый неименованным списком ввод

При управляемом неименованным списком вводе действуют правила:

- ввод выполняется из последовательных текстовых файлов, внутренних файлов или с клавиатуры;
- поле ввода содержит константу (или повторяющуюся константу), тип которой должен соответствовать элементу списка ввода, например:

```
real a, b, c
read(*, *) a, b, c
write(*, *) a, b, c                !     1.440000      1.440000      1.440000
```

Введем, например:

7*1.44

- в случае ввода числовых значений пробелы всегда обрабатываются как разделители между полями; ведущие пробелы перед первым полем записи игнорируются;
- символы конца записи имеют такое же действие, как и пробелы, за исключением случая, когда они расположены внутри символьной константы;
- допустимо использовать запятую в качестве разделителей между полями ввода;
- при наличии между полями ввода слеша (/) ввод прекращается и все последующие элементы списка ввода не изменяют своих значений.

При задании констант полей ввода следует придерживаться правил:

- *вещественные константы* одинарной или двойной точности должны быть числовыми входными полями, т. е. полями, пригодными для преобразования с использованием дескриптора F;

- *комплексные константы* являются упорядоченной парой вещественных или целочисленных констант, разделенных запятой и заключенных в круглые скобки;
- *логические константы* содержат обязательные символы T (t) или F (f), перед которыми может быть проставлена необязательная точка. Далее могут следовать необязательные символы. Так, символы T, или .t, или tru, или T1, или .t1, или .T44, или .true. могут быть использованы для представления логической константы .TRUE.;
- *символьные константы* задаются строками символов, заключенных в апострофы (') или кавычки ("). Каждый ограничитель внутри символьной константы должен быть представлен двумя одинарными ограничителями, между которыми не должно быть пробелов. Символьные константы могут быть продолжены в следующей записи. При этом символы конца записи не становятся частью символьной константы, например:

```
character(80) st
read(*, *) st
write(*, *) st                    ! Line1 - next line and last line
```

Введем:

```
'Line1
 - next line
 and last line'
```

Символьная константа может быть также задана и без ограничителей, но в таком случае константа не может включать символы-разделители: пробелы, запятые, символы конца строки, слеши. Также невозможно разместить такую константу на нескольких строках;

- если длина символьной константы меньше или равна длине вводимого элемента, то будут введены все символы константы, невведенные символы будут заполнены пробелами. Если же длина символьной константы больше длины *n* вводимого элемента, то будут введены первые *n* символов константы;
- задание производного типа выполняется путем перечисления значений для его компонентов в порядке, который задан при объявлении производного типа.

Поля ввода содержат пустые (*null*) значения, если:

- между двумя последовательными разделителями полей ввода символы не указаны, например: 11.1, , ,12.2;
- перед первым разделителем в записи символы не указаны;
- задана повторяющаяся константа с пустым значением, например задание 7* эквивалентно заданию 7 полей ввода с пустыми значениями.

Если элементу списка ввода соответствует *null*-поле, то значение элемента в результате выполнения оператора ввода не меняется.

Пробелы рассматриваются как часть разделителя за исключением:

- пробелов, встроенных в заданную с ограничителями символьную строку;
- ведущих пробелов первой записи, если только сразу после них не следует запятая или слеш (/).

Пример:

```
complex :: z = (1, 2)
real :: a = 3.3, b = 2.2
logical :: fl = .true.
character(30) :: st = 'ab'
read(*, *) z, a, b, m, n, fl, st
write(*, *) z, a, b, '\n\r'c, m, n, fl, ' ', st
```

Введем:

```
 , 1.1  , , ,  3 /
```

Результат:

```
  (1.000000,2.000000)         1.100000         2.200000
0           3   T   ab
```

9.9.2.2. Управляемый неименованным списком вывод

Вывод под управлением неименованного списка выполняется так:

- вывод осуществляется в последовательные текстовые файлы, внутренние файлы, на экран или принтер;
- длина создаваемой при выводе записи не превышает 79 символов. Если же для размещения элементов вывода требуется большее число символов, то создаются новые записи. В конце каждой записи проставляются символы конца записи: CHAR(13) и CHAR(10);
- *символьные данные* по умолчанию выводятся без ограничителей, однако после задания в операторе OPEN спецификатора DELIM = 'QUOTE' или DELIM = 'APOSTROPHE' вывод символьного значения выполняется с ограничителями: кавычками или апострофами. При этом если в символьной величине есть ограничители, то они будут удваиваться;
- вывод объекта *производного типа* выполняется покомпонентно в порядке появления компонентов в объявлении производного типа.

Управляемый неименованным списком вывод данных различных типов выполняется в соответствии с приведенными в табл. 9.8 ДП, которые, как видно из таблицы, различаются в CVF и FPS.

Таблица 9.8. Дескрипторы преобразований для вывода под управлением неименованного списка

Типы данных	*CVF*	*FPS*

LOGICAL(1), LOGICAL(2), LOGICAL(4)	L2 выводится: □T для .TRUE., □F для .FALSE.	L1 выводится: T для .TRUE., F для .FALSE.
BYTE, INTEGER(1)	I5	I11
INTEGER(2)	I7	I11
INTEGER(4)	I12	I11
REAL(4)	1PG15.7E2	F15.6 ($1 \le val < 107$) E15.6E2 ($val < 1$ или $val \ge 107$)
REAL(8)	1PG24.16E2	E24.15 ($1 \le val < 107$) E24.15E3 ($val < 1$ или $val \ge 107$)
CHARACTER(*w*)	1X, A*w*	A*w*

В таблице использованы следующие обозначения:

- □ - пробел;
- *val* - выводимая величина;
- *w* - размер символьной строки.

Пример 1. Вывод "длинной" константы:

```
character :: sub*10 = '1234567890', st*150 = ' '
do 1, i = 1, 15
1 st = trim(st) // sub
write(*, *) st
```

Пример 2. Сравнение выводов CVF и FPS:

```
character(3) :: st = 'abc'
print *, st                    ! CVF:  □abc
                               ! FPS:  abc
print '(a3)', st               ! CVF:  abc
                               ! FPS:  bc
```

Разница при форматном выводе строки объясняется тем, что в CVF по умолчанию CARRIAGECONTROL равен 'LIST', а в FPS - 'FORTRAN'.

Замечание. Для управления выводом можно использовать СИ-символы: '\n'c - новая строка, '\r'c - возврат каретки, '\t'c - табуляция и др. (разд. 3.5.5). Например:

```
character(4) year(5) /'1998', '1999', '2000', '2001', '2002'/
write(*, *) 3.55, '\t'c, 'pels', '\n\r'c, (year(i),' ', i = 1,5)
```

Результат:

```
 3.550000  pels
1998 1999 2000 2001 2002
```

10. Файлы Фортрана

10.1. Внешние и внутренние файлы

В Фортране различают два вида файлов: внешние и внутренние.

Внешний файл - файл, существующий в среде, внешней по отношению к выполняемой программе.

Внутренние файл - символьная строка (подстрока) или массив.

Внутренние файлы являются открытыми по умолчанию. Внешние файлы должны быть открыты (подсоединены к устройству В/В) оператором OPEN.

К файлам Фортрана можно организовать либо последовательный, либо прямой доступ. К некоторым видам файлов - и тот и другой. С внутренними файлами используется только последовательный доступ.

Внешние файлы могут быть:

- *форматными* (текстовыми, ASCII);
- *двоичными* (бинарными);
- *неформатными*.

Двоичный и неформатный файлы содержат неформатные записи, т. е. записи, создаваемые без преобразования данных. Файл не может одновременно содержать форматные и неформатные записи.

Внешние файлы могут быть открыты как для *монопольного*, так и для *разделенного* (сетевого) доступа. Можно создать *временный* (*scratch*) внешний файл, который будет удален с физического устройства либо после его закрытия, либо при нормальном завершении программы. При разделенном доступе внешний файл можно *заблокировать* (сделать недоступным для другого процесса).

10.2. Позиция файла

В результате выполнения операции над внешним файлом он может находиться:

- в *начальной точке файла* - непосредственно перед первой записью;
- между соседними записями файла;
- в пределах одной записи;
- в *конечной точке файла* - после последней записи до специальной записи "конец файла";
- на записи "конец файла";
- после специальной записи "конец файла".

Факт перемещения на "конец файла" устанавливается функцией EOF, которая возвращает .TRUE., если файл позиционирован в конце файла или вслед за ним, и .FALSE. - в противном случае.

Файл оказывается после записи "конец файла", если в результате выполнения оператора READ возникла ситуация "конец файла". Файл не должен быть установлен после записи "конец файла" перед началом передачи данных. Для изменения ситуации в файлах с последовательным доступом употребляются операторы REWIND или BACKSPACE.

10.3. Устройство ввода/вывода

Для передачи данных файл Фортрана подсоединяется к устройству В/В.

Устройство внешнего файла задается целочисленным скалярным выражением или звездочкой (*). Возвращаемый им результат называется *номером устройства В/В*, значение которого должно находиться в интервале от 0 до 32767.

Устройство внутреннего файла задается переменной стандартного символьного типа.

Устройство используется для ссылки на файл.

Кроме файлов к устройствам В/В могут быть подсоединены физические устройства, например клавиатура, экран, принтер, параллельный порт. Всегда в каждой Фортран-программе существуют устройства *, 0, 5 и 6. Причем по умолчанию к устройствам *, 0 и 5 подсоединена *клавиатура*, а к устройствам *, 0 и 6 - *экран*. Так, в программе

```
real :: b = 1.2
write(*, '(F6.2)') b
write(0, '(F6.2)') b
write(6, '(F6.2)') b
end
```

все операторы WRITE обеспечат вывод значения переменной *b* на экран.

Внешний файл подсоединяется к устройству В/В в результате выполнения оператора OPEN. После подсоединения и устройство и файл считаются открытыми. Доступ к внешнему файлу, после того как он открыт, выполняется по номеру устройства, к которому он подсоединен: все программные компоненты, ссылающиеся на одно и то же устройство, ссылаются на один и тот же файл. Аналогом такого номера являются в СИ указатель на файл, в Паскале - файловая переменная.

Пример:

```
integer :: k = 2, m = 4
! Устройство В/В - целочисленное скалярное выражение
open(k * m, file = 'd:\a.txt')      ! Файл d:\a.txt подсоединен к устройству 8
open(m / k, file = 'd:\b.txt')      ! Файл d:\b.txt подсоединен к устройству 2
write(8, '(i8)') k                  ! Пишем в файл d:\a.txt
write(m - k, '(i2)') m              ! Пишем в файл d:\b.txt
close(8)                            ! Закрываем устройство 8 и файл d:\a.txt
close(k)                            ! Закрываем устройство 2 и файл d:\b.txt
```

Одно и то же устройство В/В в любой программной единице выполняемой программы ссылается на один и тот же файл, например:

```
program fOpen
 integer, parameter :: n = 9
 open(n, file = 'a.txt')              ! Подсоединяем файл a.txt к устройству 9
 write(n, *) 'Test string'            ! Формируем одну запись в файле a.txt
 call ReadFileData( )                 ! Читает первую запись файла a.txt
end program fOpen

subroutine ReadFileData( )            ! В подпрограмме устройство 9 ссылается на
 character(30) :: st                  ! файл a.txt
 rewind 9                             ! Переход на начало файла a.txt
 read(9, '(a)') st                    ! Читаем первую запись файла a.txt
 print *, st                          ! Test string
end subroutine ReadFileData
```

Устройство не может быть одновременно подсоединено более чем к одному файлу, также и файл не может быть одновременно подсоединен более чем к одному устройству.

10.4. Внутренние файлы

Различают два основных типа внутренних файлов:

- символьную скалярную переменную, элемент символьного массива, символьную подстроку. Каждый такой файл имеет одну запись, длина которой совпадает с длиной образующего файл символьного элемента;
- символьный массив. Число записей такого файла совпадает с числом элементов символьного массива. Длина записи файла равна длине элемента символьного массива.

Для передачи данных во внутренний файл употребляются операторы WRITE и READ. При этом можно использовать как форматный В/В, так и В/В под управлением неименованного списка.

Перед исполнением оператора В/В внутренние файлы всегда позиционируются в начало файла. Внутренние файлы после их создания всегда открыты как для чтения, так и для записи. Устройством внутреннего файла является имя строки, подстроки, символьного массива или его элемента.

Часто внутренние файлы применяются для создания строк, содержащих смесь символьных и числовых данных (разд. 3.8.7), а также для простых преобразований "число – строка" и "строка – число", например:

```
real :: a = 234.55
integer kb
character(20) st
write(st, *) a                        ! Преобразование "число – строка"
print *, st                           !     234.550000
! Для FPS:
```

```
read(st, '(i8)') kb              ! Преобразование "строка – число"
! Для CVF:
read(st, '(i5)') kb              ! Преобразование "строка – число"
print *, kb                      !       23
end
```

10.5. Внешние файлы

Внешние файлы характеризуются приводимыми ниже понятиями.

Тип записи определяет, имеют ли записи файлов одинаковую длину, или они могут быть разной длины, или задает способ определения конца одной записи и начала другой.

Доступ к файлу определяет метод, используемый при чтении и записи данных, независимо от их организации. Способ организации файла не всегда определяет метод доступа к его записям.

CVF поддерживает два вида *организации файла*: *последовательную* и *связанную*. В CVF организация задается спецификатором ORGANIZATION оператора OPEN. Файлы, имеющие связанную организацию, сохраняются на диске, последовательную - как на диске, так и на магнитной ленте. Все иные периферийные устройства, такие, как терминалы или принтеры, рассматриваются Фортраном как файлы с последовательной организацией.

Замечание. Файлы со связанной организацией мы будем также называть *связанными файлами*.

Последовательно организованный файл состоит из записей, расположенных в порядке их поступления в файл.

В связанных файлах записи имеют одинаковую длину и хранятся в так называемых ячейках. Каждая ячейка имеет номер из диапазона [1, *n*], где *n* - номер последней доступной ячейки. *Номер ячейки* - это номер записи, исчисляемый относительно начала файла. В CVF для удаления записи связанного файла употребляется оператор DELETE.

Фортран поддерживает два метода доступа к внешнему файлу: *последовательный* и *прямой* - и 3 структуры файлов: *форматную*, *неформатную* и *двоичную*. Поэтому можно создать файлы:

- форматные последовательные;
- форматные прямые;
- неформатные последовательные;
- неформатные прямые;
- двоичные последовательные;
- двоичные прямые.

Замечание. Файлы, подсоединенные для прямого доступа, мы будем, для сокращения, также называть *прямыми файлами*, а для последовательного доступа - *последовательными*.

В последовательных файлах существующие записи могут только читаться, но не могут редактироваться. Без потери информации новые записи добавляются только после последней записи файла. Попытка вывести запись до этой записи приведет к отсечению последующих записей - их замене на добавляемую. Для изменения записи последовательного файла возможен такой путь: прочитать все записи файла в массив; изменить в нем нужную запись; перейти на начало файла и записать массив в файл.

В прямых файлах доступ к записи выполняется по номеру записи, задавая который можно читать, добавлять, замещать или удалять запись. В этом и состоит основное отличие между последовательными и прямыми файлами. При включении спецификатора REC в операторы передачи данных и управления файлами прямой файл позиционируется вслед за указанной этим спецификатором записью.

Для внешних файлов справедливо:

- для обеспечения доступа файл должен быть открыт (подсоединен к устройству В/В);
- при открытии файла по умолчанию он позиционируется на первую запись файла или на "конец файла", если в файле нет ни одной записи;
- последней записью файла является специальная запись "конец файла".

10.6. Записи

10.6.1. Типы записей

Запись - это последовательность значений (в случае неформатных и двоичных файлов) или последовательность символов (в случае форматных файлов). *Поле записи* - часть записи, содержащая данные, которые могут быть использованы оператором ввода. *Тип записи* определяет способ хранения полей в пределах записи. Тип записи не сохраняется как атрибут файла. Однако применение с файлом типа записи, отличного от используемого при создании файла, может привести в некоторых случаях к непредсказуемым результатам.

Запись является *текущей*, если файл установлен внутри записи, в противном случае текущей записи нет.

Если опущена опция компилятора /fpscomp:ioformat, в CVF доступны следующие 6 типов записей:

1) фиксированной длины. Такие записи возможны в файлах и с последовательной и со связанной организацией;
2) переменной длины; возможны только в файлах с последовательной организацией;

3) сегментированные; возможны только в файлах с последовательной организацией, открытых для неформатного последовательного доступа. Сегментированные файлы являются прерогативой CVF и не могут быть использованы другими платформами;
4) потоки без разделителей между записями; возможны только в файлах с последовательной организацией;
5) CR-потоки; используют CR (CHAR(13), carriage return) в качестве разделителя между записями; употребляются в файлах с последовательной организацией;
6) LF-потоки; используют CR и LF (CHAR(10), line feed) в качестве разделителей между записями; применяются в файлах с последовательной организацией.

10.6.2. Записи фиксированной длины

Записи фиксированной длины имеют приведенную на рис. 10.1 структуру.

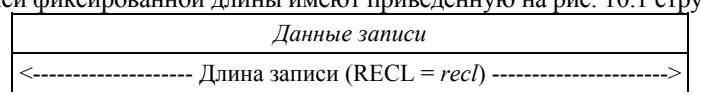


Рис. 10.1. Структура записи фиксированной длины

Записи фиксированной длины имеют связанные файлы и файлы с последовательной организацией, открытые для прямого доступа. Длина записи задается в операторе OPEN спецификатором RECL.

Пример:

```
type point
  real x, y
end type point
integer i
type(point) :: pt1 = point(1.0, 1.0), pt2 = point(2.0, 2.0), pt3 = point(3.0, 3.0)
open(1, file = 'a.txt', organization = 'sequential', access = 'direct',                &
        form = 'formatted', recordtype = 'fixed', recl = 20)
write(1, '(2f10.3)', rec = 1) pt1                ! Заносим в файл 3 записи
write(1, '(2f10.3)', rec = 2) pt2                ! Спецификатор RECORDTYPE = 'FIXED'
write(1, '(2f10.3)', rec = 3) pt3                ! может быть опущен
end
```

Результат (состав файла a.txt):

```
1.000     1.000     2.000     2.000     3.000     3.000
```

Замечание. Пример справедлив только для CVF, поскольку в операторе OPEN FPS спецификатор ORGANIZATION появляться не может.

10.6.3. Записи переменной длины

Такие записи могут содержать произвольное число байт (не превышающее максимально возможное значение). Их структура отображена на рис. 10.2.

	Данные записи	
4	<---------- Длина записи ---------->	4

Рис. 10.2. Структура записи переменной длины

Записи переменной длины могут существовать только в файлах с последовательной организацией. Каждая запись обрамляется 4-байтовыми полями, содержащими длину записи и выполняющими контрольные функции. Хранимая в этих полях величина возвращается при использовании оператора READ с дескриптором управления Q (разд. 9.8). Прочитанное значение можно затем употребить для определения размера списка В/В.

В CVF файлы с записями переменной длины обычно не используются как текстовые файлы, для которых, как правило, задается спецификатор RECORDTYPE = 'STREAM_LF'.

Пример:

```
integer recl
open(2, file = 'a.txt', recordtype = 'variable', form = 'unformatted')
! Заносим в файл 3 записи
write(2) 123, 555                          ! Длина записи 8 байт
write(2) 'Next record'                     ! Длина записи 11 байт
write(2) 1.4e-6                            ! Длина записи 4 байта
close(2)
! Закрыли файл, чтобы открыть его для форматного доступа
! и прочитать длину второй записи
open(2, file = 'a.txt', recordtype = 'variable', form = 'formatted')
! Спецификатор FORM = 'FORMATTED' может быть опущен
read(2, '(a)')                             ! Переход на начало второй записи
read(2, '(Q)') recl                        ! Читаем число байт в записи 2
print *, recl                              !    11
end
```

10.6.4. Сегментированные записи

Сегментированные записи состоят из одной или более переменной длины неформатных записей в последовательно организованном дисковом файле. В FPS таких записей нет; в CVF по умолчанию неформатные данные записываются в файлы с последовательной организацией, открытые для последовательного доступа, в виде сегментированных записей. Они полезны при работе с длинными записями, в случаях, когда нет возможности (из-за ограничений по размеру виртуальной памяти) или желания формировать одну длинную запись. Тогда она разбивается на

сегменты, которые и образуют результирующую запись. Каждый сегмент является физической записью, а результирующая запись рассматривается как единая логическая. Последняя (в случае дискового файла) может превышать максимально допустимый размер записи ($2.14*10^9$ байт), но каждый сегмент не должен быть больше этого размера.

Для доступа к сегментированному файлу задаются спецификаторы FORM = 'UNFORMATTED' и RECORDTYPE = 'SEGMENTED'. Если они не заданы и открывается ранее созданный сегментированный файл, то работа с ним может сопровождаться ошибками. Структура сегментированной записи представлена на рис. 10.3.

		Данные сегмента
2	2	<---------- Длина сегмента ---------->

Рис. 10.3. Структура сегментированной записи

Контрольная информация, предваряющая запись, содержит 4 байта. Первые два содержат длину сегмента, два последующих - идентификатор сегмента, принимающий значения:

- 1 в случае первого сегмента;
- 2 в случае последнего;
- 3 при наличии одного сегмента;
- 0 для всех промежуточных (между первым и последним) сегментов.

Если длина сегмента - нечетное число, то пользовательские данные будут увеличены на 1 байт, содержащий пробел.

10.6.5. Потоки

Поток не группируется в записи и не содержит контрольной информации. Файлы-потоки применяются с CARRIAGECONTROL = 'NONE' и содержат символьные или двоичные данные. Передаваемая порция данных и позиция файла определяются размером списка В/В. Структура потока дана на рис. 10.4.

Данные потока	EOF

Рис. 10.4. Поток

10.6.6. CR-потоки

Записи CR-потока имеют переменную длину и завершаются символом возврата каретки, который автоматически проставляется при добавлении записи в файл-поток и удаляется при ее чтении. Именно этот символ и позволяет определить длину текущей записи. Поскольку CR-потоки завершаются символом CHAR(13), в вводимых записях этот символ должен отсутствовать. Структура записи CR-потока изображена на рис. 10.5.

Данные записи	CR
<---------- Длина записи ---------->	1

Рис. 10.5. Запись CR-потока

Пример:

```
integer recl
open(3, file = 'a.txt', recordtype = 'stream_cr')
! Заносим в файл 3 записи; он открыт как текстовой (форматный) файл
! В случае вывода под управлением списка оператор WRITE вставляет
! в начало каждой записи пробел
write(3, *) 123, 555                    ! Длина записи 24 байта (см. разд. 9.9.2.2)
write(3, *) 'Next record'               ! Длина записи 13 байт
write(3, *) 1.4e-6                      ! Длина записи 16 байт
backspace(3)                            ! Возврат на одну запись
read(3, '(Q)') recl                     ! Читаем число байт в третьей записи
print *, recl                           !    16
end
```

Результат (состав файла a.txt):

```
   123         555
 Next record
  1.4000000E-06
```

10.6.7. LF-потоки

Записи LF-потока имеют переменную длину и завершаются символами возврата каретки и новой строки, которые автоматически проставляются при добавлении записи и удаляются при чтении. Эти символы позволяют определить длину текущей записи. Поскольку LF-потоки завершаются символами CHAR(13) и CHAR(10), в вводимых записях эти символы должны отсутствовать. LF-потоки - это стандартные текстовые файлы. Структура записи LF-потока представлена на рис. 10.6.

Данные записи	CR	LF
<---------- Длина записи ---------->	1	1

Рис. 10.6. Запись LF-потока

Пример:

```
integer recl
open(4, file = 'a.txt', recordtype = 'stream_lf')
! Заносим в файл 3 записи; он открыт как текстовой (форматный) файл
write(4, '(2i5)') 123, 555              ! Длина записи 10 байт
write(4, '(a30)') 'Next record'         ! Длина записи 30 байт
```

```
write(4, '(e20.8)') 1.4e-6                ! Длина записи 20 байт
rewind(4)                                 ! Переход на начало файла
read(4, '(Q)') recl                       ! Читаем число байт в третьей записи
print *, recl                             !    10
end
```

Результат (состав файла a.txt):

```
 123  555
           Next record
   0.14000000E-05
```

10.7. Передача данных с продвижением и без

Оператор В/В с продвижением всегда устанавливает файл после последней считанной или записанной записи (если не было ошибок).

Оператор В/В без продвижения, применяемый при форматной передаче данных и задаваемый спецификатором ADVANCE = 'NO' или (при выводе) дескрипторами \ и $, может устанавливать файл внутри текущей записи. Используя такой оператор, можно прочитать или записать одну запись с помощью нескольких операторов В/В, причем каждый из них будет обращаться к части записи, например:

```
character(6) :: st1, st2
open(1, file = 'a.txt')
write(1, *) 'Test string'                 ! Вывод с продвижением
rewind 1                                  ! Переход на начало файла
read(1, '(a6)', advance = 'no') st1       ! Ввод без продвижения
print *, st1                              ! Test
read(1, '(a6)', advance = 'no') st2       ! Ввод второго слова в ту же запись
print *, st2                              ! string
```

10.8. Позиция файла перед передачей данных

Порядок установки позиции файла перед передачей данных зависит от способа доступа к файлу.

В случае последовательного доступа перед *вводом*, если есть текущая запись, позиция файла не меняется. Иначе файл устанавливается на начало следующей записи и она становится текущей. Ввод запрещен, если нет следующей записи или следующей является запись "конец записи". Такая ситуация в файле с последовательным доступом возникнет, когда последним обратившимся к нему оператором является оператор WRITE.

Перед *выводом* в файл с последовательным доступом, если есть текущая запись, то позиция файла не меняется и текущая запись становится последней записью файла. Иначе, например если файл находился перед записью "конец файла", создается новая запись, в которую будут передаваться данные и которая становится текущей и последней записью файла. Позиция файла устанавливается на начало этой записи.

При прямом доступе перед передачей данных позиция файла устанавливается в начало записи, определяемой спецификатором оператора B/B REC. Эта запись становится текущей.

10.9. Позиция файла после передачи данных

При *вводе без продвижения*, если не было ситуации ошибки или ситуации "конец файла", но есть ситуация "конец записи", позиция файла устанавливается после только что считанной записи. Если же в операторе ввода без продвижения не было ситуаций ошибки, "конец файла" или "конец записи", то позиция файла не меняется. В операторе *вывода без продвижения*, если не было ситуации ошибки, позиция файла не изменяется.

Во всех остальных случаях файл устанавливается вслед за последней считанной или записанной записью, так что она становится предшествующей записью.

10.10. Двоичные последовательные файлы

При работе с двоичными файлами обмен данными выполняется без их преобразования. При записи в двоичный файл в него фактически копируется содержимое ячеек оперативной памяти. При чтении, наоборот, последовательность байтов двоичного файла передается в ячейки оперативной памяти, отведенные под элементы ввода. В FPS число записей в двоичном файле равно числу переданных байт. Поэтому выполнение оператора BACKSPACE приведет к перемещению на 1 байт назад. Последовательный двоичный файл является потоком. Между его записями CVF и FPS не проставляют символов или полей с контрольными данными.

Ввод из двоичного файла значения переменной *val*, занимающей в оперативной памяти *n* байт, вызовет перемещение файла на *n* байт.

Если последовательный двоичный файл перед выполнением оператора вывода находился на байте b_i, то b_i и все последующие байты в результате вывода будут "затерты" (заменены на выводимые).

Оператор OPEN, подсоединяющий файл к устройству для двоичного последовательного доступа, обязательно включает спецификатор FORM = *form*, где *form* - символьное выражение, вычисляемое со значением 'BINARY'.

Пример управления двоичным последовательным файлом FPS:

```
integer(2) :: ia, ib, d(5) = (/ 1, 2, 3, 4, 5 /), i
real(4) :: a
character(3) ca
open(1, file = 'a.dat', form = 'binary')   ! Открываем двоичный файл
write(1) 1.1, 2.2                          ! Пишем в файл 8 байт
write(1) d                                 ! Добавляем в файл 10 байт
write(1) 'aaa', 'bbb', 'ccc'               ! Добавим в файл еще 9 байт
```

```
rewind 1
read(1) a, a, (ia, i = 1, 5), ca, ca      ! Читаем 24 байта одним оператором READ
do 2 i = 1, 10                            ! Перемещение назад на 10 записей (байт)
2 backspace 1
read(1) ib                                ! Читаем, начиная с 15-го байта
write(*, *) a, ia, ca, ib                 !    2.200000    5bbb    4
rewind 1
read(1) a                                 ! Файл переместился на 5-й байт
write(1) 'ghi'                            ! Заменены 5, 6 и 7-й байты;
rewind 1                                  ! все последующие записи "затерты"
read(1) a, ca
write(*, *) a, ca                         !       1.100000ghi (в файле 7 байт данных)
end
```

Замечания:

1. В CVF при работе с двоичными файлами оператор BACSPACE либо вызывает ошибку исполнения, либо, подобно оператору REWIND, позиционирует файл в его начало, например:

```
character(1) c
open(1, file = 'a.dat', form = 'binary')  ! Открываем двоичный файл
write(1) 'a12', 'b34'                     ! Добавим в файл 6 байт
rewind 1                                  ! Переход на начало файла
read(1) c
print *, 'c = ', c                        ! CVF:    c = a
                                          ! FPS:    c = a
rewind 1                                  ! Переход на начало файла
! Те же действия, но с оператором BACKSPACE
write(1) 'a12', 'b34'                     ! Добавим в файл 6 байт
backspace 1
read(1) c
print *, 'c = ', c                        ! CVF:    c = a
end                                       ! FPS:    c = 4
```

2. Любой внешний файл может быть открыт как двоичный (поток), например с целью копирования данных.

10.11. Неформатные последовательные файлы

В неформатные файлы, так же как и в двоичные, данные передаются без преобразований. Однако в неформатном файле в отличие от двоичного записью является не байт, а совокупность данных, выводимых в файл в результате выполнения оператора вывода WRITE. Каждый оператор вывода создает одну запись. Записи файла могут иметь разную длину. Каждая запись завершается символами конца записи.

Из неформатного файла одним оператором ввода можно прочитать только одну запись. Попытка такого ввода числа байт, превышающих

размер текущей записи, приведет к ошибке выполнения и прерыванию программы.

Выполнение каждого оператора ввода, даже если число вводимых байт меньше числа байт записи, приведет к позиционированию файла вслед за прочитанной записью.

Добавление новой записи при позиционировании файла вслед за записью r_{i-1} приведет к удалению записи r_i и всех последующих записей (к их замене на добавляемую).

Оператор OPEN, подсоединяющий файл к устройству для неформатного последовательного доступа, включает спецификатор FORM = *form*, где *form* - символьное выражение, вычисляемое со значением 'UNFORMATTED'.

Пример управления неформатным последовательным файлом:

```
integer(2) :: ia, ib, d(4) = (/ 1, 2, 3, 4 /)
real(4) a
character(3) ca                 ! Открываем неформатный файл
open(1, file = 'a.dat', form = 'unformatted')
write(1) 1.1, 2.2               ! Добавление в файл 1-й записи
write(1) d                      ! Вторая запись
write(1) 'aaa', 'bbb', 'ccc'    ! Третья запись
rewind 1                        ! Переход в начало файла
read(1) a, a                    ! Соблюдаем соответствие между
read(1) ia, ia, ia, ia          ! вводом и выводом
read(1) ca, ca
backspace 1                     ! Переход в начало 3-й записи
backspace 1                     ! Переход в начало 2-й записи
read(1) ib, ib
print *, a, ia, ca, ib          !     2.200000    4bbb    2
rewind 1
read(1) a                       ! Переход в начало 2-й записи
write(1) 'ghi'                  ! Замена 2-й и 3-й записей на ghi
rewind 1
read(1) a, a                    ! В файле осталось 2 записи:
read(1) ca                      ! числа 1.1 и 2.2 и строка ghi
print *, a, ca                  ! 2.200000ghi
end
```

10.12. Текстовые последовательные файлы

Текстовой файл содержит символьное представление данных всех типов. При работе с текстовыми файлами используется форматный или управляемый списком В/В. При *выводе* данные из внутреннего представления преобразовываются во внешнее символьное представление. При *вводе* происходит обратное преобразование из символьного представления во внутреннее (разд. 9.1).

При *выводе* в конце каждой записи Фортран проставляет два неотображаемых символа CHAR(13) - возврат каретки и CHAR(10) - новая строка,. То есть записи являются LF-потоками. В случае вывода под управлением списка оператор вывода вставляет в начало каждой записи пробел (по умолчанию первый символ каждой записи форматного файла рассматривается как символ управления кареткой). Записи текстового последовательного файла могут иметь разную длину.

Порядок изменения позиции текстового файла с последовательным доступом зависит от способа передачи данных (с продвижением или без, разд. 10.6).

При форматном вводе число читаемых одним оператором ввода из текущей записи данных регулируется форматом (разд. 9.4). В отличие от неформатного файла одним оператором ввода в принципе может быть прочитано произвольное число записей текстового последовательного файла.

В простейшем случае, открывая текстовой файл для последовательного форматного доступа, можно указать в операторе OPEN только устройство внешнего файла и спецификатор FILE = *file*.

Пример управления текстовым последовательным файлом. (По умолчанию в файлах, открываемых для последовательного доступа, спецификатор FORM = 'FORMATTED'.):

```
integer(2) :: ia, ib, d(4) = (/ 1, 2, 3, 4 /)
real(4) a, b
character(3)ca
1 format(6f7.2)
2 format(8i5)
3 format(7a4)
! Открываем последовательный текстовой файл a.txt и создаем в нем 3 записи
open(10, file = 'a.txt')
write(10, 1) 1.1, 2.2                ! Запись 1
write(10, 2) d                       ! 2
write(10, 3) 'a', 'bc', 'def'        ! и 3
! После выполнения трех операторов вывода в файле будет 3 записи:
!   1.10   2.20
!    1    2    3    4
!   a  bc def
rewind 10                            ! Переход на 1-ю запись
read(10, 1) a, b
read(10, 2) ia, ia, ia               ! Читаем из 2-й записи
! или  вместо двух последних операторов: read(1, *) a, b, ia, ia, ia
read(10, 3) ca, ca                   ! Читаем из 3-й записи
backspace 10                         ! Переход на начало 3-й записи
backspace 10                         ! Переход на начало 2-й записи
read(10, 2) ib, ib                   ! Читаем в ib 2-й элемент 2-й записи
write(*, *) a, ia, ca, ib            !    1.100000   3   bc   2
rewind 10                            ! Переход на начало 1-й записи
```

```
read(10, *)                    ! Переход на начало 2-й записи
write(10, 3) 'ghij'            ! Все записи, начиная со 2-й, заменены на ghij
! После выполнения трех последних операторов имеем файл:
!   1.10   2.20
! ghij
end
```

10.13. Файлы, подсоединенные для прямого доступа

Для прямого доступа можно открыть двоичные, неформатные и форматные (текстовые) файлы. В файле, подсоединенном для прямого доступа, все записи имеют одинаковую длину, задаваемую при открытии файла спецификатором RECL. При этом в случае неформатного и текстового файлов число байт, передаваемых одним оператором В/В, не должно превышать длины записи. В случае вывода недостающие байты записи будут содержать *null*-символы.

Необязательно читать или записывать записи по порядку их номеров. В подсоединенный к устройству файл можно занести любую запись. Например, можно записать запись 3, даже если в файле нет записей с номерами 1 и 2. При этом, однако, между началом файла и записью 3 будет зарезервировано пространство для записей 1 и 2:

```
open(1, file = 'a.txt', form = 'formatted', access = 'direct', recl = 7, status = 'new')
write(1, '(i7)', rec = 3) 12          ! Добавляем запись 3
```

Запрещается передавать записи при помощи форматирования под управлением списка (именованного и неименованного).

В файле с прямым доступом можно позиционироваться непосредственно вслед за записью r_i, выполнив оператор READ, в котором задан спецификатор REC = r_i. Список ввода такого оператора может быть пуст.

Позиционирование вслед за записью r_i в неформатном или двоичном файле:

```
read(2, rec = ri)
```

и в форматном файле:

```
read(2, '(a)', rec = ri)           ! Дескриптор формата - любой
```

В CVF для позиционирования прямого файла можно также употребить оператор FIND (разд. 11.8).

В файле с прямым доступом при выполнении оператора WRITE будет обновлена та запись, номер которой задан спецификатором REC оператора WRITE. В FPS, если спецификатор REC в операторе WRITE отсутствует, то обновляется текущая запись. Все последующие записи файла будут сохранены. Если же указанная спецификатором REC запись не существует, то она будет добавлена в файл.

Замечание. При работе с прямыми файлами спецификатор REC может отсутствовать только в FPS; для CVF он обязателен.

В FPS, чтобы удалить ненужные завершающие записи прямого файла, следует переместиться вслед за последней сохраняемой записью (это обычно выполняется оператором READ), а затем применить оператор ENDFILE. Этот способ в CVF неприменим, поскольку в нем ENDFILE работает только с последовательными файлами.

Оператор OPEN, подсоединяющий файл *file* для неформатного прямого доступа к устройству *u*, должен иметь спецификаторы:

OPEN(*u*, FILE = *file*, ACCESS = 'direct', RECL = *recl*)

где *recl* - целочисленное выражение, возвращающее длину записи файла.

Спецификатор FORM = 'UNFORMATTED' в случае прямого файла задается по умолчанию и может быть опущен. В случае форматного и двоичного файла с прямым доступом спецификатор FORM является обязательным:

OPEN(*u*, FILE = *file*, ACCESS = 'direct', FORM = 'formatted', RECL = *recl*)

OPEN(*u*, FILE = *file*, ACCESS = 'direct', FORM = 'binary', RECL = *recl*)

Пример для FPS. В прямом файле a.dat, содержащем 30 записей, удалить записи, имеющие номер, больший 15.

```
character(30) :: fn = 'a.dat'
character(35) :: st = 'One line'
integer :: ios, r = 15, ner, i
! Сначала создадим файл из 30 записей
open(1, file = fn, access = 'direct', form = 'formatted', recl = 35)
endfile 1                              ! На тот случай, если файл существует
! Установим файл перед только что проставленной записью "конец файла"
rewind 1
write(1, '(a35)') (st, i = 1, 30)      ! Теперь в файле 30 записей
close(1)
open(2, file = fn, access = 'direct', form = 'formatted',                    &
        recl = 35, status = 'old', iostat = ios)
if(ios .ne. 0) stop 'Cannot open file a.dat'
read(2, '(a)', rec = r, iostat = ios)  ! Переход в начало записи 16
if(ios .eq. 0) then
  ! Если удалось прочитать запись r, то проставляем метку конца файла
  endfile 2                            ! В CVF оператор ENDFILE употребляется
else                                   ! только с последовательными файлами
 write(*, *) 'Не могу прочитать запись ', r
end if
inquire(2, nextrec = ner)              ! ner - номер следующей записи
print *, ner                           !     16
rewind 2                               ! Выполним контрольный вывод
k = 0
```

```
do while(.not. eof(2))
 k = k + 1                          ! Номер читаемой записи
 read(2, '(a)', rec = k) st
 print *, st, k                     ! Выведено 15 записей
end do
end
```

Замечания:

1. Код неприемлем для CVF, поскольку в нем оператор ENDFILE применим только с файлами, открытыми для последовательного доступа.

2. При работе в CVF с файлами с прямым доступом использованный в примере циклический список

```
write(1, '(a35)') (st, i = 1, 30)
```

неприменим, поскольку в CVF при записи в прямой файл оператор WRITE должен содержать спецификатор REC. Поэтому используется цикл

```
do i = 1, 30                        ! Цикл приемлем и в FPS и в CVF
  write(1, '(a35)', rec = i) st
end do
```

3. В файле с прямым доступом CVF в отличие от FPS не проставляет символ новой строки после каждой записи прямого файла. Эта разница иллюстрируется примером:

```
character(30) :: fn = 'a.dat'
character(8) :: st = 'A record'
open(1, file = fn, access = 'direct', form = 'formatted', recl = 9, status = 'new')
write(1, '(a8)', rec = 1) st
write(1, '(a8)', rec = 2) st
write(1, '(a8)', rec = 3) st
end
```

Состав файла a.dat:

```
CVF:  A record A record A record
FPS:  A record
      A record
      A record
```

4. Чтобы установить прямой файл вслед за записью *r*, можно использовать оператор READ без списка ввода: READ(2, '(A)', REC = *r*), а в CVF - также оператор FIND, например: FIND(2'*r*).

5. При работе с текстовыми прямыми файлами возможен только форматный В/В. Передача данных под управлением списка недопустима. Также невозможен и В/В без продвижения.

Файл с записями, переданными в него в режиме прямого доступа, можно впоследствии открыть как двоичный с последовательным доступом (как поток).

Пример. В файл, открытый для прямого доступа с RECL = 15, заносятся 3 записи. Затем этот же файл открывается как двоичный для последовательного доступа и данные побайтно переносятся в форматный файл b.txt. Вывод файла b.txt выполняется после его подсоединения с RECL = 15.

```
integer(4), parameter :: n = 15
character(1) c
character(n) st
integer i
100 format(a<n>)                    ! или: 100 format(a15)
open(1, file = 'a.txt', access = 'direct', form = 'formatted',                    &
       organization = 'relative', recl = n)
! Создаем в прямом файле 3 записи
write(1, 100, rec = 1) 'First record'
write(1, 100, rec = 2) 'Second record'
write(1, 100, rec = 3) 'Last record'
close(1)                            ! Закрываем файл a.txt
! Файл-источник
open(1, file = 'a.txt', form = 'binary')
! Файл-приемник
open(2, file = 'b.txt', form = 'formatted', recordtype = 'fixed', recl = 1)
do while(.not. eof(1))              ! Копируем данные посимвольно
  read(1) c                         ! Читаем 1 байт из источника
  write(2, '(a)') c                 ! Пишем 1 байт в приемник
end do
close(1); close(2)                  ! Отсоединяем файлы a.txt и b.txt
! Откроем теперь b.txt с RECL = n и прочитаем 3 записи
open(2, file = 'b.txt', access = 'direct', form = 'formatted', recl = n)
i = 0                               ! Номер читаемой записи
do while(.not. eof(2))              ! Копируем данные посимвольно
  i = i + 1
  read(2, 100, rec = i) st
  print *, st                       !       First record
end do                              !    Second record
end                                 !       Last record
```

Отличия между неформатным и двоичным файлами с прямым доступом:

- в двоичный файл можно записывать любое число байт, не обращая внимания на значение спецификатора RECL (при этом, правда, длина записи все же определяется спецификатором RECL);
- из двоичного файла одним оператором ввода можно считать больше байтов, чем задано спецификатором RECL.

Один и тот же двоичный или неформатный файл может быть открыт с разными значениями спецификатора RECL.

Пример:

```
character(14) ch
integer ner
open(2, file = 'a.bin', access = 'direct', form = 'binary', recl = 8)
write(2, rec = 1) 'C12-', '92'     ! Пишем в файл 8 байт
write(2, rec = 2) 'C16-', '99'     ! Вторая запись двоичного файла
! Читаем 14 байт из двоичного файла, т. е. больше, чем задано RECL
read(2, rec = 1) ch
inquire(2, nextrec = ner)
print *, ner                       !     3
write(*, *) ch                     ! C12-92  C16-99
end                                ! Прочитать 14 байт из неформатного файла,
                                   ! открытого с RECL = 8, нельзя
```

Приведенный текст в FPS может выглядеть так:

```
character(14) ch
integer ner
open(2, file = 'a.bin', access = 'direct', form = 'binary', recl = 8)
write(2) 'C12-', '92'              ! Пишем в файл 8 байт
write(2) 'C16-', '99'              ! Вторая запись двоичного файла
rewind 2
read(2) ch                         ! Читаем 14 байт из двоичного файла
inquire(2, nextrec = ner)
print *, ner                       !     3
write(*, *) ch                     ! C12-92  C16-99
end
```

В CVF этот код неприменим, поскольку, во-первых, операторы B/B не содержат спецификатор REC, а во-вторых, использован оператор REWIND, который в CVF употребляется только с последовательными файлами.

В FPS текстовой прямой файл устроен так же, как и текстовой последовательный файл, в котором все записи имеют одинаковую длину. Поэтому текстовой файл с равными по длине записями может быть открыт в FPS как для прямого, так и для последовательного доступа.

Пример. В текстовом файле, подсоединенном для прямого доступа, каждая запись имеет поля "Фамилия И. О.". По ошибке в некоторых записях фамилия начинается со строчной буквы. Исправить ошибку в исходном файле и сформировать отчет в виде двоичного файла, содержащего исправленные записи.

```
integer(4), parameter :: n = 15
type person
  character(len = n) lastn, firstn
end type person
```

```
type(person) line
integer(4) :: code, dco, cco, ut = 2, ub = 3, ir
100 format(2a15)
open(ut, file = 'a.txt', form = 'formatted', access = 'direct', recl = 2 * n)
open(ub, file = 'b.dat', form = 'binary')
! Создадим две записи
! Запись без ошибки
write(ut, fmt = 100, rec = 1) person('Blake', 'William')
! Запись с ошибкой
write(ut, fmt = 100, rec = 2) person('maugham', 'W. Somerset')
code = ichar('Z')                    ! 90 - ASCII-код символа Z
dco = ichar('z') - code              ! Вернет dco = 32
ir = 0
do
 ir = ir + 1                         ! Номер текущей записи в прямом файле
 read(ut, fmt = 100, rec = ir) line  ! или: read(ut, '(3a)', rec = ir) line
 cco = ichar(line.lastn(1:1))
 if(cco > code) then                 ! Если первая буква фамилии строчная, то:
  line.lastn(1:1) = char(cco - dco)  ! • переводим строчную букву в прописную;
  write(ut, fmt = 100, rec = ir) line ! • исправляем ошибку в исходном файле;
  write(ub) line                     ! • пишем в двоичный файл
 end if
 if(eof(ut)) exit                    ! Если ситуация "конец файла"
end do
close(ut)                            ! Закрываем прямой файл
rewind ub                            ! Контрольный вывод
do while(.not. eof(ub))
 read(ub) line
 write(*, *) line
end do
close(ub)
end
```

Результат. Файл a.txt *после исправлений* (случай CVF; в случае FPS каждая запись размещается на отдельной строке):

```
Blake          William          Maugham          W. Somerset
```

Пояснение. Строчные буквы в таблице ASCII расположены после прописных. Для получения ASCII-кода прописной буквы по известному коду строчной достаточно вычесть из кода строчной буквы *dco*:

```
dco = ichar('z') - ichar('Z').
```

10.14. Удаление записей из файла с прямым доступом

Приводимые в разделе сведения применимы только в FPS, поскольку в обсуждаемом механизме присутствует оператор ENDFILE, неупотребляемый в CVF с прямыми файлами.

Способ удаления завершающих записей из файла прямого доступа FPS был рассмотрен в предыдущем разделе.

Удаление ненужных промежуточных записей можно выполнить, пользуясь методом, принятым в системах управления базами данных.

Во-первых, необходимо иметь возможность отмечать удаляемые записи (а также снимать эту отметку). Для этой цели в записи можно выделить отдельное, однобайтовое поле, проставляя в него 1 (.TRUE.), если запись подлежит удалению, или 0 (.FALSE.), если нет. Далее необходимо будет написать процедуру, которая может работать, например, по следующему алгоритму:

- обменять помеченные для удаления записи с записями, имеющими наибольшие номера;
- установить файл перед первой помеченной для удаления записью и выполнить оператор ENDFILE.

Запуск такой процедуры следует выполнять по мере необходимости, памятуя, что удаленные записи не могут быть восстановлены.

10.15. Выбор типа файла

Двоичные и неформатные файлы имеют очевидные преимущества перед текстовыми:

- передача данных выполняется быстрее, поскольку нет потерь на преобразование данных;
- в текстовых файлах из-за округления могут возникнуть потери точности;
- программировать двоичный и неформатный В/В значительно проще, чем форматный;
- двоичные и неформатные файлы, как правило, имеют меньший размер, чем текстовые файлы с теми же данными.

Последнее обстоятельство проиллюстрируем примером:

```
real(4) :: a(20) = 1255.55
open(1, file = 'a.txt')
write(1, '(5f8.2)') a                    ! Размер текстового файла 168 байт
open(2, file = 'a.dat', form = 'binary')
write(2) a                               ! Размер двоичного файла 80 байт
end
```

Программа создает два файла. В текстовом файле a.txt под данные занято 160 байт и дополнительно 8 байт (2*4) займут символы конца записи. Всего в файле a.txt будет создано 4 записи, каждая запись будет содержать 5 полей длиной по 8 символов. Размер двоичного файла a.dat составит 80 байт: всего в файл будет выведено 20 элементов по 4 байта каждый. То есть в случае двоичного файла имеем существенную экономию внешней памяти.

Текстовые файлы содержат данные в пригодной для чтения форме и также используются для обмена данными между программами, работающими в различных операционных системах. Примером такого рода обменных файлов являются DXF-файлы программы Автокад.

Выбор способа доступа к файлу (последовательный или прямой) определяется характером решаемых задач. Если необходимо редактировать записи файла, или индексировать файл по одному или нескольким полям записи, или делать недоступными отдельные записи файла для других процессов, то используются файлы прямого доступа.

11. Операции над внешними файлами

Внешний файл доступен из программы, если он, во-первых, открыт и, во-вторых, не заблокирован другим процессом. Файл открывается в результате подсоединения его к устройству В/В, которое, в свою очередь, создается (открывается) оператором OPEN. Столь же равнозначно можно говорить и о подсоединении устройства к файлу. Между устройством *u* и файлом *file* существует однозначное соответствие: все операторы Фортрана в любой программной единице, ссылающиеся на устройство *u*, получают доступ к файлу *file*. К устройству может быть подсоединен как существующий, так и вновь создаваемый файл. Нельзя подсоединить один и тот же файл к разным устройствам одновременно, так же как и нельзя одновременно подсоединить одно устройство к разным файлам.

После того как файл открыт, с ним возможны операции:

- позиционирования (BACKSPACE, REWIND, ENDFILE, READ, WRITE);
- передачи данных (READ, WRITE, PRINT и ENDFILE);
- изменения свойств подсоединения (оператор OPEN);
- опроса (INQUIRE и EOF).

Закрывается файл в результате выполнения оператора CLOSE или при нормальном завершении программы.

Управляющие файлами и выполняющие их опрос операторы перечислены в табл. 11.1. Их подробное описания дано в последующих разделах.

Таблица 11.1. Применяемые при работе с файлами операторы

Оператор	*Назначение*
Операторы, применимые в CVF и FPS	
BACKSPACE	Возвращает файл на одну запись назад
REWIND	Позиционирует файл в начало его первой записи
ENDFILE	Записывает специальную запись "конец файла"
OPEN	Создает устройство В/В и подсоединяет к нему внешний файл
CLOSE	Отсоединяет файл от устройства В/В и закрывает это устройство
READ	Выполняет передачу данных из файла, подсоединенного к устройству В/В, в указанные в списке ввода переменные
WRITE	Передает данные из списка вывода в файл, подсоединенный к устройству В/В
PRINT	Выводит данные на экран (устройство *)
INQUIRE	Возвращает свойства устройства или внешнего файла

Функция EOF	Возвращает .TRUE., если подсоединенный к устройству В/В файл позиционирован на специальной записи "конец файла" или после этой записи. Иначе EOF возвращает .FALSE.
Операторы, применимые в CVF	
ACCEPT	Выполняет форматный ввод данных или ввод под управлением списка клавиатуры
TYPE	Синоним PRINT; все правила для PRINT переносятся на TYPE
REWRITE	Замещает текущую запись на новую в файлах прямого доступа
FIND	Позиционирует прямой файл на заданную запись
DELETE	Удаляет заданную запись связанного файла
UNLOCK	Освобождает запись файла, закрытую для доступа предшествующим оператором READ

Приведенные операторы (кроме PRINT), а также функция EOF содержат в качестве одного из спецификаторов устройство, к которому подсоединяется файл в результате выполнения оператора OPEN. Этот спецификатор является обязательным. В то же время каждый из работающих с файлами операторов имеет и необязательные, рассматриваемые в настоящей главе спецификаторы. Целесообразность их употребления определяется как решаемыми задачами, так и требованиями к надежности программного продукта.

При описании операторов их необязательные элементы заключаются в квадратные скобки. Символ | используется для обозначения "или".

Замечание. В CVF операторы BACKSPACE, REWIND и ENDFILE употребляются только с файлами, открытыми для последовательного доступа, т. е. так, как это предусмотрено стандартом Фортрана. В FPS эти операторы можно вдобавок использовать с файлами прямого доступа.

11.1. Оператор BACKSPACE

Оператор возвращает файл на одну запись назад и имеет две формы:

BACKSPACE *u*

и

BACKSPACE([UNIT =] *u* [, ERR = *err*] [, IOSTAT = *iostat*])

u - устройство внешнего файла (разд. 10.2).

err - метка исполняемого оператора. При возникновении ошибки В/В управление передается на оператор, имеющий метку *err*.

iostat - целочисленная переменная, принимающая значение 0 при отсутствии ошибок. В противном случае *iostat* равняется номеру ошибки.

При отсутствии подсоединения возникнет ошибка выполнения.

Оператор BACKSPACE позиционирует файл ровно на одну запись назад, кроме приведенных в табл. 11.2 случаев.

Таблица 11.2. Специальные случаи позиционирования файла оператором BACKSPACE

Случай	*Результат*
Нет предшествующей записи	Положение файла не меняется
Предшествующая запись - "конец файла"	Файл позиционируется до записи "конец файла"
Файл находится в пределах одной записи	Файл перемещается в начало этой записи

В FPS в файлах, содержащих вывод, выполненный под управлением именованного списка, BACKSPACE рассматривает список как множество записей, число которых равно числу элементов списка плюс две записи: одна - заголовок списка, другая - завершающий список слеш (/).

Пример:

```
integer :: lunit = 10, ios
backspace 5
backspace(5)
backspace lunit
backspace(unit = lunit, err = 30, iostat = ios)
```

Замечания:

1. В CVF оператор BACKSPACE употребляется только с файлами, подсоединенными для последовательного доступа. Вдобавок он не используется с записями, созданными под управлением списка, как именованного, так и без имени.

2. Если спецификатор UNIT= опущен, то параметр *u* должен быть первым параметром оператора. В противном случае параметры могут появляться в произвольном порядке. Это замечание справедливо для *всех* приводимых в этой главе операторов, имеющих спецификатор UNIT=.

3. Если параметром оператора является выражение, которое содержит вызов функции, то эта функция не должна выполнять В/В или функцию EOF. В противном случае результаты непредсказуемы. Это замечание распространяется на *все* приводимые в этой главе операторы.

11.2. Оператор REWIND

Оператор перемещает файл в начало первой записи файла.

REWIND *u*

или

REWIND([UNIT =] *u* [, ERR = *err*] [, IOSTAT = *iostat*])

Описание параметров *u*, *err* и *iostat* см. в разд. 11.1.

При отсутствии подсоединения оператор REWIND не вызывает никаких действий.

Замечание. В CVF оператор REWIND употребляется только с файлами, подсоединенными для последовательного доступа.

11.3. Оператор ENDFILE

Оператор записывает специальную запись "конец файла".

ENDFILE *u*

или

ENDFILE([UNIT =] *u* [, ERR = *err*] [, IOSTAT = *iostat*])

Описание параметров *u*, *err* и *iostat* см. в разд. 11.1.

Если файл не открыт, то возникает ошибка выполнения.

После выполнения записи "конец файла" файл позиционируется вслед за этой записью. Дальнейшая последовательная передача данных станет возможной только после выполнения операторов BACKSPACE или REWIND. После выполнения оператора ENDFILE все записи, расположенные после новой записи "конец файла", становятся недоступными. В FPS это справедливо как для последовательных файлов, так и для прямых файлов.

По понятным причинам ENDFILE нельзя применять с файлами, открытыми с ACTION | MODE = 'READ'.

Замечание. В CVF оператор ENDFILE употребляется только с файлами, подсоединенными для последовательного доступа.

11.4. Оператор OPEN

Оператор OPEN создает устройство В/В с номером *u* и подсоединяет к нему внешний файл *file*. При успешном подсоединении файл считается открытым и к нему может быть обеспечен доступ других работающих с файлами операторов Фортрана. Оператор может быть использован и для подсоединения ранее открытого файла с целью изменения свойств подсоединения. Отмеченные звездочкой спецификаторы оператора могут применяться только в CVF.

```
OPEN([UNIT =] u [, ACCESS = access] [, ACTION = action]                &
  [, ASSOCIATEVARIABLE = asv*] [, BLANK = blank]                       &
  [, BLOCKSIZE = blocksize] [, BUFFERCOUNT = bc*]                      &
  [, BUFFERD = bf*] [, CARRIAGECONTROL = carriagecontrol]              &
  [, CONVERT = fm*] [, DEFAULTFILE = def*] [, DELIM = delim]           &
  [, DISPOSE = dis*] [, ERR = err] [, FILE = file] [, FORM = form]
```

```
[, IOFOCUS = iofocus] [, IOSTAT = iostat] [, MAXREC = mr*]      &
[, MODE = mode] [, ORGANIZATION = org*] [, PAD = pad]           &
[, POSITION = position] [, READONLY*] [, RECL = recl]           &
[, RECORDTYPE = rtyp*] [, SHARE = share] [, SHARED*]            &
[, STATUS = status] [, TITLE = title]                           &
[, USEROPEN = user-function-name*])
```

u - устройство внешнего файла (разд. 10.2), к которому подсоединяется файл *file*.

access - символьное выражение, вычисляемое со значениями 'APPEND', 'DIRECT' или 'SEQUENTIAL' (по умолчанию) и определяющее способ доступа к файлу (последовательный - 'SEQUENTIAL' или прямой - 'DIRECT'). Спецификатор ACCESS = 'APPEND' применяется при работе с последовательными файлами, открываемыми для добавления данных. При успешном выполнении оператора OPEN с ACCESS = 'APPEND' файл позиционируется перед записью "конец файла".

action - символьное выражение, задающее возможные действия с файлом и вычисляемое со значениями 'READ' (процесс может только читать данные из файла), 'WRITE' (возможен только вывод данных) или 'READ-WRITE' (можно и читать и записывать данные).

Если спецификатор ACTION не задан, то система пытается открыть файл для чтения-записи ('READWRITE'). Если попытка неуспешна, то система пытается открыть файл снова первоначально только для чтения ('READ'), затем только для записи ('WRITE').

Значение спецификатора STATUS не оказывает никакого влияния на *action*.

asv - целочисленная переменная стандартного целого типа, называемая *ассоциируемой переменной файла*; обновляется после каждой передачи данных при работе с прямыми файлами и содержит номер следующей записи файла. Имеет эффект только в CVF с операторами READ, WRITE, FIND, DELETE и REWRITE. Например:

```
integer(4) asv                      ! Ассоциируемая переменная файла
character(8) :: st = 'A record'
! Создадим прямой файл с тремя записями
open(1, file = 'c.dat', access = 'direct', form = 'formatted', recl = 9,        &
     status = 'new', associatevariable = asv)
write(1, '(a8)', rec = 1) st; write(1, '(a8)', rec = 2) st; write(1, '(a8)', rec = 3) st
print *, asv                        !      4
read(1, '(a8)', rec = 1) st
print *, asv                        !      2
```

blank - символьное выражение, вычисляемое со значениями 'NULL' или 'ZERO'. В случае 'NULL' (устанавливается по умолчанию) пробелы при форматном вводе данных игнорируются (вводится в действие дескриптор BN). В случае 'ZERO' пробелы при форматном вводе данных рассматриваются как нули (вводится в действие дескриптор BZ). Однако

если одновременно заданы параметр *blank* оператора OPEN и дескрипторы BN или BZ в спецификации формата оператора B/B, то дескриптор формата перекрывает действие параметра *blank*.

blocksize - выражение стандартного целого типа, задающее размер внутреннего буфера в байтах.

bc - скалярное целочисленное выражение, задающее число буферов, ассоциируемых с устройством B/B при многобуферной передаче данных. Возвращаемое выражением значение должно находиться в диапазоне от 1 до 127. По умолчанию задействован один буфер. Спецификатор BLOCKSIZE задает размер одного буфера. Общее число байт, ассоциируемых с заданными буферами, если, например, BLOCKSIZE = 2048 и BUFFERCOUNT = 3, равно 3 * 2048 = 6144.

bf - символьное выражение, вычисляемое со значением 'YES' или 'NO', определяющее характер передачи данных. Если *bf* возвращает 'NO', то данные в файл будут посылаться после каждого выполнения оператора WRITE. В противном случае данные, если позволяет физическое устройство и вид файла, предварительно накапливаются во внутреннем буфере, что может повысить производительность приложения. По умолчанию размер буфера - 8192 байта, а в случае FPS - 1024 байта. Общий размер буфера может быть изменен спецификаторами BLOCKSIZE и BUFFERCOUNT. Внутренний буфер может увеличиваться, чтобы разместить целиком запись, и никогда не уменьшается.

carriagecontrol - символьное выражение, задающее способ интерпретации первого символа каждой записи в форматных файлах. Выражение может вычисляться со значениями 'FORTRAN' или 'LIST'. По умолчанию устройство *u* подсоединяется к внешнему устройству, например к принтеру или монитору, с *carriagecontrol* = 'FORTRAN'. Это означает, что первый символ записи интерпретируется как символ управления кареткой печатающего устройства и не выводится ни на принтер, ни на экран (разд. 9.1). К внешним файлам по умолчанию подсоединение выполняется с *carriagecontrol* = = 'LIST'.

В случае 'LIST' первый символ записи уже не интерпретируется как символ управления кареткой и выводится и на принтере, и на экране.

Если в OPEN также задан спецификатор FORM = 'UNFORMATTED' или FORM = 'BINARY', то спецификатор CARRIAGECONTROL игнорируется.

fm - скалярное символьное выражение, задающее вид представления числовых неформатных данных и вычисляемое со значением 'LITTLE ENDIAN', 'BIG ENDIAN', 'CRAY', 'FDX', 'FGX', 'IBM', 'VAXD', 'VAXG' или 'NATIVE'. Используется для приведения данных к соответствующему виду.

def - скалярное символьное выражение, задающее используемый по умолчанию путь к открываемому файлу. Если завершающий слеш (/) опущен, то он будет добавлен. Если спецификатор DEFAULTFILE отсутствует, то используется текущая рабочая директория.

delim - скалярное символьное выражение, задающее ограничитель для символьных данных при В/В под управлением именованного или неименованного списка. Выражение может вычисляться со значениями 'APOSTROPHE', 'QUOTE' или 'NONE' (по умолчанию). Если ограничитель задан, то внутренние, совпадающие с ограничителем символы строки (апостроф (') или кавычки (")) удваиваются (разд. 9.9.1.2).

dis - скалярное символьное выражение, задающее статус файла после его отсоединения от устройства. Принимает одно из следующих значений:

- 'KEEP' или 'SAVE' - файл сохраняется;
- 'DELETE' - файл удаляется;
- 'PRINT' - файл подается на печать и сохраняется (только для последовательных файлов);
- 'PRINT/DELETE' - файл подается на печать и удаляется (только для последовательных файлов);
- 'SUBMIT' - расщепляет процесс для исполнения файла;
- 'SUBMIT/DELETE' - расщепляет процесс для исполнения файла и удаляет его после завершения операций.

По умолчанию действует статус 'DELETE' для *scratch*-файлов и 'KEEP' - для всех остальных. Вторая форма спецификатора - DISP = *dis*.

err - метка исполняемого оператора. При возникновении ошибки управление передается на оператор, имеющий метку *err*.

file - символьное выражение, задающее имя файла, подсоединяемого к устройству с номером *u*. Если спецификатор FILE не задан, то создается временный, стираемый после выполнения оператора CLOSE или после нормального завершения программы файл. При этом должен быть задан спецификатор STATUS со значением SCRATCH, например:

```
open(1, status = 'scratch', form = 'bynary')
```

Если параметр *file* вычисляется со значением *пробел*, то выполняются следующие действия:

- программа читает имя файла из списка аргументов (если таковые имеются) в командной строке, запускающей программу. Если аргументом является нулевая строка (''), то имя файла будет предложено ввести пользователю. Каждый последующий оператор OPEN, имеющий в качестве имени файла пробел, читает соответствующий аргумент командной строки;
- если операторов OPEN, имеющих в качестве имени файла пробел, больше, чем аргументов командной строки, программа потребует ввести недостающие имена файлов.

Если имя файла 'USER' или 'CON', то вывод выполняется на экран, ввод - с клавиатуры. Имя *file* может задавать и другие физические устройства, например принтер (FILE = 'PRN') или первый последовательный порт

(FILE = = 'COM1'). В приложениях QuickWin задание FILE = 'USER' позволяет открыть дочернее окно. Тогда все операции В/В, связанные с устройством, к которому это окно подсоединено, выполняются на это окно.

form - символьное выражение, вычисляемое со значениями 'FORMATTED', 'UNFORMATTED' или 'BINARY'. Если доступ к файлу последовательный, то по умолчанию устанавливается форма 'FORMATTED'. Если доступ прямой, то по умолчанию устанавливается форма 'UNFORMATTED'.

iofocus - логическое выражение, в случае истинности которого дочернее окно приложения QuickWin устанавливается в фокусе (располагается поверх других окон) при выполнении операторов READ, WRITE и PRINT. Является расширением над стандартом Фортран 90.

iostat - целочисленная переменная, возвращающая 0 при отсутствии ошибок, отрицательное число, если возникла ситуация "конец файла", или номер возникшей ошибки.

mr - скалярное выражение, задающее максимальное число записей, которое может быть передано при работе с прямым файлом в период его подсоединения к устройству. При необходимости преобразовывается в целый тип. По умолчанию число передаваемых записей не ограничено.

mode - символьное выражение, задающее подобно *action* возможные действия с файлом и вычисляемое со значениями 'READ' (процесс может только читать данные из файла), 'WRITE' (возможен только вывод данных в файл) или 'READWRITE' (возможен как ввод, так и вывод данных). Является расширением над стандартом Фортрана.

org - скалярное символьное выражение, задающее внутреннюю организацию файла, вычисляемое со значением 'SEQUENTIAL', если задается файл с последовательной организацией (действует по умолчанию), или 'RELATIVE', если задается файл со связанной организацией.

pad - символьное выражение, вычисляемое со значениями 'YES' (по умолчанию) или 'NO'. В случае 'YES' если при форматном вводе требуется больше данных, чем содержится в записи, то недостающее число данных восполняется пробелами. Если же PAD = 'NO', то при попытке форматного ввода большего числа данных, чем содержится в записи, возникнет ошибка ввода. Например:

```
character(20) :: st
open(1, file = 'a.txt', pad = 'yes')
read(1, '(a)') st                    ! Читаем из файла и выводим на экран
print *, st                          ! abcd
read *                               ! Ждем нажатия Enter
! Изменяем свойство подсоединения PAD файла a.txt
open(1, file = 'a.txt', pad = 'no')  ! По умолчанию POSITION = 'ASIS'
read(1, '(a)') st                    ! Возникнет ошибка ввода
```

Файл a.txt:

abcd
efgh

position - символьное выражение, задающее способ позиционирования файла при последовательном доступе и которое должно вычисляться со значениями 'ASIS', 'REWIND' или 'APPEND'. Если имеет место 'REWIND', то существующий файл позиционируется в начало файла. В случае 'APPEND' существующий файл позиционируется непосредственно перед записью "конец файла". В случае 'ASIS' (задается по умолчанию) позиция ранее подсоединенного файла не изменяется, в то время как ранее неподсоединенный файл позиционируется в свое начало.

READONLY – так же как и при задании ACTION = 'READ', файл подсоединяется только для чтения. Однако READONLY предотвращает удаление файла, если присутствует спецификатор STATUS = 'DELETE'.

recl - целочисленное выражение, задающее длину каждой записи в байтах. Этот параметр задается только в файлах прямого доступа.

rtyp - скалярное символьное выражение, задающее тип записей файла и вычисляемое с одним из следующих значений (см. также разд. 10.5):

- 'FIXED' - записи фиксированной длины;
- 'VARIABLE' - записи переменной длины;
- 'SEGMENTED' - сегментированные записи;
- 'STREAM' - поток;
- 'STREAM_LF' - LF_поток;
- 'STREAM_CR' - CR_поток.

При подсоединении файла действуют умолчания:

- 'FIXED' - для связанных файлов и файлов с последовательной организацией, открытых для прямого доступа;
- 'STREAM_LF' - для форматных файлов с последовательной организацией;
- 'VARIABLE' - для неформатных файлов с последовательной организацией.

share - символьное выражение, вычисляемое со значениями 'DENYRW', 'DENYWR', 'DENYRD' или 'DENYNONE':

- 'DENYRW' - (deny-read/write mode) пока файл открыт в этом режиме, никакой другой процесс не может открыть этот файл ни для чтения, ни для записи;
- 'DENYWR' - (deny-write mode) пока файл открыт в этом режиме, никакой другой процесс не может открыть этот файл для записи;
- 'DENYRD' - (deny-read mode) пока файл открыт в этом режиме, никакой другой процесс не может открыть этот файл для чтения;
- 'DENYNONE' - (deny-none mode) пока файл открыт в этом режиме, любой другой процесс может открыть этот файл как для чтения, так и для записи.

SHARED - подсоединяемый файл открывается для разделенного доступа, при котором к файлу могут обращаться более одного приложения.

status - символьное выражение, которое может принимать значения 'OLD', 'NEW', 'SCRATCH', 'REPLACE' или 'UNKNOWN':

- 'OLD' - файл должен уже существовать, в противном случае возникнет ошибка В/В;
- 'NEW' - файл не должен существовать. Если он не существует, то он будет создан, в противном случае возникнет ошибка В/В;
- 'SCRATCH' - если в операторе OPEN опущен параметр *file*, то по умолчанию значение *status* равно 'SCRATCH'. Создаваемые 'SCRATCH'-файлы являются временными и уничтожаются либо при закрытии устройства, либо при завершении программы;
- 'REPLACE' - открываемый файл замещает существующий файл с тем же именем. Если такого файла не существует, то создается новый файл;
- 'UNKNOWN' (по умолчанию) - процесс прежде пытается открыть файл со статусом 'OLD', затем - со статусом 'NEW'. Если файл существует, то он открывается, если нет - создается.

Значение *status* затрагивает только дисковые файлы и игнорируется при работе с устройствами.

Пример:

```
character(70) fn
write(*, '(a\)') 'Введите имя файла: '
read(*, '(a)') fn
! Открываем неформатный новый файл прямого доступа
! Файл fn должен отсутствовать на диске
open(7, file = fn, access = 'direct', status = 'new')
```

Подсоединение файла к устройству *звездочка* (*) не имеет никакого действия, поскольку это устройство постоянно связанно с клавиатурой и экраном. Однако можно подсоединить внешний файл к существующим по умолчанию устройствам 0, 5 и 6.

Если в операторе OPEN использовано устройство, подсоединенное ранее к другому файлу, то ранее открытый файл автоматически закрывается, а затем другой файл открывается и подсоединяется к заданному параметром *u* устройству. Нельзя одновременно подсоединить один и тот же файл к разным устройствам.

Если файл не открыт и выполняется оператор READ или WRITE, то программа попытается открыть файл так, как это делает оператор OPEN, в котором задан параметр FILE = ' '.

Если оператор OPEN подсоединяет устройство к несуществующему файлу, то файл открывается со свойствами, заданными в операторе.

Можно использовать оператор OPEN, указывая в нем имя уже подсоединенного файла с тем же устройством для изменения свойств

подсоединения, задаваемых спецификаторами BLANK, DELIM, PAD, ERR и IOSTAT. В таких случаях спецификатор POSITION = 'APPEND' игнорируется, но спецификатор POSITION = 'REWIND' вызывает перемещение на начало файла.

Пример:

```
integer(4) :: k
open(7, file = 'a.txt', blank = 'zero')
write(7, '(2i3)') 1, 2
rewind 7
read(7, '(i6)') k                    ! Возвращает 1002
! Изменяем свойство подсоединения BLANK файла a.txt
open(7, file = 'a.txt', blank = 'null')
rewind 7
read(7, '(i6)') k                    ! Возвращает 12
end
```

title - символьное выражение, задающее имя дочернего окна QuickWin. Если спецификатор TITLE задан в приложении, которое не является QuickWin, то возникнет ошибка выполнения.

USEROPEN = *function name* - спецификатор, позволяющий передать управление внешней функции *function name*, непосредственно открывающей файл. Вызываемая функция должна иметь атрибут EXTERNAL, открывать файл, используя функцию CreateFile, и возвращать дескриптор файла, вырабатываемый CreateFile; используется, когда нужно задать не предусмотренные оператором OPEN свойства подсоединения. Основное назначение спецификатора - использовать возможности WIN32 API функции CreateFile, находящейся, кстати, в библиотеке kernel32.lib.

Пример. Приводимый код можно употребить для вызова заданной спецификатором USEROPEN функции. При выполнении оператора OPEN передается управление функции *fileopen*, которая вызывает функцию CreateFile. Необходимые для работы CreateFile данные собраны в модуле MTY, интерфейс CreateFile размещен в *fileopen*.

```
character(30) :: st, fn
integer(4), external :: fileopen
fn = 'a.dat'c                        ! СИ-строка, завершаемая null-символом
open(1, file = 'a.dat', status = 'old', useropen = fileopen, err = 10)
write(1, *) 'Test string'
rewind 1                             ! Переход на начало файла
read(1, '(a)') st
print *, st                          ! Test string
stop 'OK'                            ! Файл удачно открыт
10 print *, 'Error'
end
```

```
module mty                          ! Вместо модуля DFWINTY
 ! Константа file_flag_write_through взята из файла dfwinty.f90
 integer, parameter :: file_flag_write_through = #80000000
 type t_security_attributes         ! Взят из файла dfwinty.f90
    sequence
    integer(4) :: nLength, lpSecurityDescriptor
    logical(4) :: bInheritHandle
   end type t_security_attributes
end module mty

! Возможный вид функции fileopen
integer function fileopen(lpFileName, dwDesiredAccess, dwShareMode,              &
        lpSecurityAttributes, dwCreationDisposition, dwFlagsAndAttributes,       &
        hTemplateFile, unit)
 !dec$attributes reference :: dwDesiredAccess
 !dec$attributes reference :: dwShareMode
 !dec$attributes reference :: dwCreationDisposition
 !dec$attributes reference :: dwFlagsAndAttributes
 !dec$attributes reference :: unit
 use mty                            ! Вместо USE DFWINTY
 ! Интерфейс CreateFile взят из kernel32.f90. Однако в оригинале lpFileName
 ! имеет тип CHARACTER*(*), что неверно. Ниже ошибка исправлена
 interface
  integer(4) function CreateFile(lpFileName, dwDesiredAccess, dwShareMode,       &
        lpSecurityAttributes, dwCreationDisposition, dwFlagsAndAttributes,       &
        hTemplateFile)
   !dec$ attributes default :: CreateFile
   !dec$ attributes stdcall, alias : '_CreateFileA@28' :: CreateFile
   !dec$ attributes reference :: lpFileName
   !dec$ attributes reference :: lpSecurityAttributes
   use mty                          ! Вместо USE DFWINTY
   integer(4) :: lpFileName         ! Правильное объявление типа
   integer :: dwDesiredAccess, dwShareMode
   type(t_security_attributes) lpSecurityAttributes
   integer :: dwCreationDisposition, dwFlagsAndAttributes, hTemplateFile
  end function CreateFile
 end interface
 integer(4) :: lpFileName, dwDesiredAccess, dwShareMode, lpSecurityAttributes,   &
        dwCreationDisposition, dwFlagsAndAttributes, hTemplateFile, unit
 type(t_security_attributes), pointer :: null_sec_attr
 ! Битовый флаг записи file_flag_write_through для CreateFile
 dwFlagsAndAttributes = dwFlagsAndAttributes + file_flag_write_through
 ! Открывает файл с помощью CreateFile
 fileopen = CreateFile(lpFileName, dwDesiredAccess, dwShareMode,                 &
        null_sec_attr, dwCreationDisposition, dwFlagsAndAttributes, hTemplateFile)
end function fileopen
```

Первые 7 параметров функции передаются из CVF и соответствуют списку параметров CreateFile. Их значения берутся из соответствующего оператора OPEN:

lpFileName - адрес СИ-строки, содержащей имя файла;

dwDesiredAccess - желаемый доступ;

dwShareMode - вид доступа (монопольный или разделяемый);

lpSecurityAttributes - в Фортране всегда *null*; передается для ссылки на структуру *security_attributes*;

dwCreationDisposition - вид выполняемых с файлом действий;

dwFlagsAndAttributes - атрибуты файла и его флаги;

hTemplateFile - всегда *null*; передается для дескриптора временного файла функции CreateFile.

Последний параметр - устройство, указанное в операторе OPEN.

11.5. Оператор CLOSE

Оператор отсоединяет файл от устройства В/В и закрывает это устройство.

CLOSE([UNIT =] *u* [, ERR = *err*] [, IOSTAT = *iostat*] &
[, STATUS | DISPOSE | DISP = *status*])

Описание параметров *u*, *err* и *iostat* см. в разд. 11.1.

status - символьное выражение, вычисляемое с одним из следующих значений:

- 'KEEP' или 'SAVE' - файл сохраняется;
- 'DELETE' - файл удаляется;
- 'PRINT' - файл подается на печать и сохраняется (только для последовательных файлов);
- 'PRINT/DELETE' - файл подается на печать и удаляется (только для последовательных файлов);
- 'SUBMIT' - расщепляет процесс для исполнения файла;
- 'SUBMIT/DELETE' - расщепляет процесс для исполнения файла и удаляет его после завершения операций.

По умолчанию действует статус 'DELETE' для *scratch*-файлов и окон QuickWin. Для остальных файлов действует 'KEEP'.

Задание STATUS = 'KEEP' для временных файлов вызывает ошибку выполнения. Для других видов файлов по умолчанию принимается статус 'KEEP'.

Если к устройству не был подсоединен файл, то никакой ошибки не возникает.

Открытые файлы не обязательно закрывать оператором CLOSE. При нормальном завершении программы они закрываются автоматически в соответствии с заданными для них статусами. Закрытие устройства (файла) 0 автоматически пересоединяет это устройство к клавиатуре и экрану.

Закрытие устройств 5 и 6 пересоединяет эти устройства соответственно к клавиатуре и экрану. Оператор CLOSE(*) вызовет ошибку компиляции.

Пример:

```
! Закрываем устройство 7 и удаляем с диска подсоединенный к нему файл
close(7, status = 'delete')
```

11.6. Оператор READ

Оператор выполняет передачу данных из подсоединенного к устройству *u* файла в указанные в списке ввода переменные. В CVF в случае прямого файла READ изменяет ассоциируемую переменную файла, указанную в соответствующем операторе OPEN. Передача данных выполняется до тех пор, пока не выполнены все операции ввода либо не возникли ситуации "конец файла" или ошибки. В случае ввода под управлением списка ввод прекращается при обнаружении в поле ввода слеша (/).

При вводе из файла или с клавиатуры оператор имеет вид:

```
READ([UNIT = ] u [, [[FMT = ] fmt] | [[NML = ] nml]            &
  [, ADVANCE = advance] [, END = end] [, EOR = eor]            &
  [, ERR = err] [, IOSTAT = iostat] [, REC = rec] [, SIZE = size]) [iolist]
```

При работе с клавиатурой оператор можно записать так:

```
READ * | fmt [, iolist]
```

Если спецификатор UNIT= опущен, то параметр *u* должен быть первым параметром оператора. Если опущены спецификаторы FMT или NML, то параметры *fmt* или *nml* должны быть вторыми параметрами оператора. В противном случае параметры могут появляться в произвольном порядке.

u - устройство В/В (разд. 10.2 и 10.3). Устройство может быть задано звездочкой (*). В таком случае ввод будет выполняться с клавиатуры.

Если устройство не было подсоединено к файлу, то при чтении будут выполнены действия, задаваемые оператором:

```
OPEN(u, FILE = ' ', STATUS = 'OLD',                              &
  ACCESS = 'SEQUENTIAL', FORM = form)
```

где *form* вычисляется со значениями 'FORMATTED' (при форматном вводе) и 'UNFORMATTED' (при неформатном). Если имя файла включено в запускающую программу командную строку, то это имя будет использовано для имени файла. В противном случае программа попросит ввести имя файла с клавиатуры.

fmt - спецификатор формата, которым может быть либо метка оператора FORMAT, либо символьное выражение, содержащее заключенный в круглые скобки список дескрипторов преобразований. При управляемом списком вводе в качестве *fmt* используется звездочка (*). Управляемый списком ввод возможен только из последовательных текстовых файлов. При неформатном или двоичном вводе параметр *fmt* должен быть опущен.

nml - спецификатор именованного списка. При вводе именованного списка *iolist* должен быть опущен. Управляемый именованным списком ввод может быть выполнен только из текстовых файлов, открытых для последовательного доступа.

advance - символьное выражение, позволяющее задать продвигающийся или непродвигающийся последовательный форматный ввод и вычисляемое со значениями 'YES' или 'NO'. Значение 'YES' задается по умолчанию и означает, что задан продвигающийся ввод, т. е. после выполнения каждого оператора ввода файл позиционируется вслед за записью, из которой выполнялась передача данных. При непродвигающемся В/В (ADVANCE = = 'NO') файл оставляется сразу за последним переданным символом.

end - метка исполняемого оператора того же блока видимости, где применен оператор READ. Если спецификатор END присутствует, то при достижении конца файла управление передается на исполняемый оператор, метка которого *end*. Внешний файл устанавливается за записью "конец файла". Если END отсутствует и не заданы спецификаторы ERR или IO-STAT, то чтение после записи "конец файла" приведет к ошибке выполнения.

eor - метка оператора того же блока видимости, в котором размещен оператор READ. Если спецификатор EOR= задан, то также должен быть задан и спецификатор ADVANCE='NO'. Если спецификатор EOR= задан, выполнение оператора ввода прекращается при достижении конца записи (если ранее не было иной ошибки). Если EOR= опущен, то при достижении конца записи возникает ошибка, которую можно обработать спецификатором IOSTAT.

err - метка исполняемого оператора. При возникновении ошибки В/В управление передается на оператор, имеющий метку *err*.

iostat - целочисленная переменная, возвращает 0 при отсутствии ошибок; возвращает -1, если возникла ситуация "конец файла"; в противном случае возвращает номер возникшей ошибки. Состояние ошибки возникает, например, если обнаружен конец записи при непродвигающемся вводе.

rec - целочисленное выражение, возвращающее положительное число, называемое *номером записи*. Спецификатор REC может быть задан только при работе с файлами прямого доступа (иначе возникнет ошибка В/В). Если REC задан, то до ввода данных файл позиционируется на начало записи с номером *rec*, что обеспечивает передачу данных из этой записи. Первая запись файла имеет номер 1. В FPS по умолчанию значение *rec* равно номеру текущей записи файла. И если при вводе из прямого файла параметр отсутствует, то будет введена текущая запись файла. В CVF при работе с прямыми файлами наличие спецификатора REC обязательно.

size - целочисленная переменная стандартного целого типа, возвращающая число переданных при выполнении форматного ввода полей с данными. Добавляемые в результате выполнения спецификатора

PAD = 'YES' пробелы не засчитываются. Спецификатор SIZE= может быть задан только при задании спецификатора ADVANCE = 'NO'. Например:

```
integer i, isv
real a(10)
open(1, file = 'a.txt')
do i = 1, 5
  read(1, '(f5.1)', advance = 'no', size = isv) a(i)
end do
print *, isv                          !      5
end
```

Файл a.txt:

```
1.0  2.0  3.0  4.0  5.0  6.0  7.0  8.0  9.0 10.0
```

iolist - список ввода, содержащий переменные, значения которых должны быть переданы из файла. Элементами списка ввода могут быть как объекты любых типов, включая и производный, так и их подобъекты.

Если при вводе возникает ошибка, то выполнение оператора прекращается, все элементы списка ввода становятся неопределенными, а положение файла непредсказуемым.

Оператор READ может нарушить выполнение некоторых графических текстовых процедур, например SETTEXTWINDOW, которая изменяет текущую позицию курсора. Чтобы этого избежать, можно в графическом режиме выполнять ввод с клавиатуры, используя функцию GETCHARQQ, и выводить результаты на экран посредством процедуры OUTTEXT.

11.7. Оператор ACCEPT

Выполняет ввод данных с клавиатуры.

```
ACCEPT fmt [, iolist]        ! Форматный ввод
ACCEPT * [, iolist]          ! Ввод под управлением списка
ACCEPT nml                   ! Ввод под управлением именованного списка
```

fmt - спецификатор формата, задаваемый без спецификатора FMT.
iolist - список ввода.
nml - спецификатор именованного списка.

Пример:

```
real x
integer k
print *, 'Enter real x and integer k'
accept *, x, k                    ! Введем: 123.45   56
print '(f7.2, i5)', x, k          ! 123.45   56
end
```

11.8. Оператор FIND

Оператор устанавливает прямой файл на запись *rec* и заносит в ассоциируемую переменную файла номер *rec*. Имеет две формы:

FIND([UNIT =] *u*, REC = *rec* [, ERR = *err*] [, IOSTAT = *iostat*])

и

FIND(*u*'*rec* [, ERR = *err*] [, IOSTAT = *ios*])

Описание спецификаторов см. в разд. 11.6.

11.9. Оператор DELETE

Удаляет запись *rec* в связанном файле.

DELETE([UNIT =] *u* [, REC = *rec*] [, ERR = *err*] [, IOSTAT = *iostat*])

Описание спецификаторов см. в разд. 11.6.

Оператор логически удаляет заданную запись из файла, помечая ее как удаленную, и освобождает эту запись для занесения новых данных. При удалении записи, если не возникло ситуации ошибки, ассоциируемая переменная файла получает значение *rec* + 1. Если спецификатор REC не задан, то удаляется текущая запись.

Пример для FIND и DELETE:

```
integer(4) ios, asv                  ! asv - ассоциируемая переменная
character(8) :: st = 'A record'
! Создадим прямой файл с тремя записями
open(1, file = 'c.dat', access = 'direct', form = 'formatted', recl = 9,&
       status = 'new', associatevariable = asv, organization = 'relative')
write(1, '(a8)', rec = 1) st; write(1, '(a8)', rec = 2) st; write(1, '(a8)', rec = 3) st
find(1, rec = 2)                     ! или: find(1'2)
print *, asv                         !      2
delete(1, rec = 2)
print *, asv                         !      3
read(1, rec = 2, iostat = ios) st
if(ios /= 0) then
  print *, 'Error'
else
  print *, st                        ! Напечатает пустую строку
end if
end
```

11.10. Оператор UNLOCK

Освобождает запись в связанном файле или в файле с последовательной организацией, заблокированную предшествующим оператором READ. Имеет две формы:

UNLOCK *u*

и

UNLOCK([UNIT =] *u* [, ERR = *label*] [, IOSTAT = *iostat*])

Описание спецификаторов см. в разд. 11.6.

Если нет заблокированной записи, оператор игнорируется.

11.11. Оператор WRITE

Оператор передает данные из списка вывода в файл, подсоединенный к устройству *u*. В CVF в случае прямого файла изменяет ассоциируемую переменную файла, указанную в соответствующем операторе OPEN.

```
WRITE([UNIT =] u [, [[FMT =] fmt] | [[NML =] nml]          &
  [, ADVANCE = advance] [, ERR = err]                      &
  [, IOSTAT = iostat] [, REC = rec]) [iolist]
```

u - устройство В/В (разд. 10.2 и 10.3). Вывод будет выполняться на экран, если в качестве устройства использована звездочка (*).

Если устройство не было подсоединено к файлу, то при выводе будут выполнены действия, задаваемые оператором:

```
OPEN(u, FILE = ' ', STATUS = 'UNKNOWN',                    &
  ACCESS = 'SEQUENTIAL', FORM = form)
```

где *form* вычисляется со значениями 'FORMATTED' (при форматном вводе) и 'UNFORMATTED' (при неформатном). Если имя файла включено в запускающую программу командную строку, то это имя будет использовано для имени файла. В противном случае программа попросит ввести имя файла с клавиатуры.

fmt - спецификатор формата. При неформатном или двоичном вводе параметр *fmt* должен быть опущен. Вывод будет управляться списком, если в качестве *fmt* использована звездочка. Управляемый списком вывод возможен только в последовательные текстовые файлы.

nml - спецификатор именованного списка. При выводе именованного списка *iolist* должен быть опущен. Управляемый именованным списком вывод может быть выполнен только в файлы, открытые для последовательного доступа.

advance - символьное выражение, позволяющее задать продвигающийся или непродвигающийся последовательный форматный вывод и вычисляемое со значениями 'YES' или 'NO'. Значение 'YES' задается по умолчанию и означает, что задан продвигающийся вывод, т. е. после выполнения каждого оператора вывода проставляются символы конца записи и файл позиционируется вслед за проставленными символами. При непродвигающемся В/В (ADVANCE='NO') символы конца записи не проставляются и файл оставляется вслед за последним выведенным символом.

err - метка исполняемого оператора. При возникновении ошибки В/В управление передается на оператор, имеющий метку *err*.

iostat - целочисленная переменная, возвращающая 0 при отсутствии ошибок или номер возникшей ошибки.

rec - целочисленное выражение, возвращающее положительное число, называемое номером записи. Параметр *rec* может быть задан только при работе с файлами прямого доступа (иначе возникнет ошибка В/В). Параметр *rec* указывает на запись, в которую будут переданы данные при выполнении оператора WRITE. В FPS по умолчанию значение *rec* равно номеру текущей записи файла. И если при вводе из прямого файла параметр отсутствует, то будет изменена текущая запись файла. В CVF при работе с прямыми файлами наличие спецификатора REC обязательно.

iolist - список вывода, содержащий выражения, результаты которых должны быть переданы в файл.

При записи в последовательный файл все записи, расположенные после введенной, удаляются (при наличии таковых) и файл позиционируется перед записью "конец файла". Таким образом, после вывода в последовательный файл необходимо применить BACKSPACE или REWIND для выполнения оператора READ. Препятствий для применения оператора WRITE, однако, нет.

11.12. Оператор PRINT

Выводит данные на экран (устройство *).

PRINT * | *fmt* [, *iolist*]

где *fmt* - спецификатор формата; *iolist* - список вывода. Если звездочка замещает *fmt*, то вывод управляется списком *iolist*.

11.13. Оператор REWRITE

Замещает в файле прямого доступа текущую запись на новую. В случае прямого файла изменяет ассоциируемую переменную файла, указанную в соответствующем операторе OPEN. При выводе в форматный файл имеет вид:

REWRITE([UNIT =] *u*, [FMT =] *fmt* [, IOSTAT = *iostat*] &
[, ERR = *err*]) [*iolist*]

С неформатными файлами употребляется так:

REWRITE([UNIT =] *u* [, IOSTAT = *iostat*] [, ERR = *err*]) [*iolist*]

Спецификаторы имеют тот же смысл, что и в операторе WRITE.

Текущей является запись, на которую установился файл в результате выполнения предшествующего оператора READ. Причем с устройством *u* между READ и REWRITE не должны выполняться другие операторы В/В,

кроме INQUIRE. В противном случае текущая запись окажется неопределенной.

Число символов в списке вывода и задаваемых спецификатором формата не должно превышать длины записи, определяемой спецификатором RECL оператора OPEN. Если же число передаваемых символов меньше длины записи, то недостающие символы восполняются пробелами.

Пример:

```
type person
  character(len = 15) lastn, firstn
end type person
integer ios
100 format(2a15)
open(3, file = 'a.txt', form = 'formatted', access = 'direct', recl = 30)
! Создадим две записи
write(3, fmt = 100, rec = 1) person('Blake', 'William')
write(3, fmt = 100, rec = 2) person('Maugham', 'W. Somerset')
read(3, fmt = 100, rec = 1)          ! Текущей является запись 1
rewrite(3, 100, iostat = ios) person('Byron', 'G. Gordon')
if(ios /= 0) print *, ' Rewriting error'
end
```

11.14. Оператор INQUIRE

Возвращает свойства устройства или внешнего файла.

Форма оператора опроса файла:

INQUIRE(FILE = *file* [, ERR = *err*] [, IOSTAT = *iostat*] &
[, DEFAULTFILE = *def*], *slist*)

Форма оператора опроса устройства:

INQUIRE([UNIT =] *u* [, ERR = *err*] [, IOSTAT = *iostat*], *slist*)

Форма оператора опроса списка вывода:

INQUIRE(IOLENGTH = *iolength*) [*iolist*]

file - символьное выражение, задающее имя файла, информацию о котором возвращает оператор INQUIRE.

u - устройство внешнего файла (разд. 10.2), о котором необходимо получить информацию. Если задано UNIT = *, то нельзя включать спецификатор NUMBER.

В операторе INQUIRE можно задать либо параметр *u*, либо *file*, но не одновременно и то и другое. Если задан параметр *u*, то выполняется опрос устройства. Если задан *file*, то выполняется опрос файла.

def - скалярное символьное выражение, задающее используемый по умолчанию путь к открываемому файлу. Если завершающий слеш (/) опущен, то он будет добавлен. Если спецификатор DEFAULTFILE отсутствует, то используется текущая рабочая директория.

Если в INQUIRE присутствует спецификатор DEFAULTFILE, то он должен быть и в соответствующем операторе OPEN. Спецификатор DEFAULTFILE = *def* может быть задан в дополнение или вместо спецификатора FILE = *file*. И *file* и *def* могут начинаться с тильды (~).

iolength - переменная стандартного целого типа, возвращающая размер списка вывода. Эта форма оператора INQUIRE включает только спецификатор IOLENGTH= и список вывода *iolist*. Все другие спецификаторы должны отсутствовать. Список *iolist* во всех других случаях отсутствует. Например:

```
real :: r = 1.1, a(100) = 2.2
integer :: iol, kar(50) = 5
character(25) :: st(25) = 'abcd'
inquire(iolength = iol) r, a, kar, st          ! iol - размер списка вывода
print *, iol                                   !       1229
```

Полученное значение можно использовать, например, для задания спецификатора RECL оператора OPEN. Затем данные можно передать в открытый неформатный файл прямого доступа.

slist - один или более спецификаторов из следующего списка:

```
[, ACCESS = access] [, ACTION = action] [, BINARY = binary]                    &
[, BLANK = blank] [, BLOCKSIZE = blocksize]                                    &
[, BUFFERD = bf*] [, CARRIAGECONTROL = carriagecontrol]                        &
[, CONVERT = fm*] [, DELIM = delim] [, DIRECT = direct]                        &
[, ERR = err] [, EXIST = exist] [, FORM = form]                                &
[, FORMATTED = formatted] [, IOFOCUS = iofocus]                                &
[, IOSTAT = iostat] [, MODE = mode] [, NAME = name]                            &
[, NAMED = named] [, NEXTREC = nextrec] [, NUMBER = num]                       &
[, OPENED = opened] [, ORGANIZATION = org*] [, PAD = pad]                      &
[, POSITION = position] [, READ = read] [, READWRITE = readwrite]              &
[, RECL = recl] [, RECORDTYPE = rtyp*] [, SEQUENTIAL = seq]                    &
[, SHARE = share] [, UNFORMATTED = unformatted] [, WRITE = write])
```

Замечание. Отмеченные звездочкой параметры применимы только в CVF.

access - символьная переменная. Возвращает 'APPEND', если заданное устройство или файл открыты для добавления данных. Возвращает 'SEQUENTIAL', если устройство или файл открыты для последовательного доступа, и возвращает 'DIRECT', если устройство или файл открыты для прямого доступа. Возвращает 'UNDEFINED' при отсутствии подсоединения.

action - символьная переменная, возвращающая 'READ', если файл открыт только для чтения, или 'WRITE', если файл открыт только для записи, или 'READWRITE', если файл подсоединен как для чтения, так и для записи. Возвращает 'UNDEFINED' при отсутствии подсоединения.

binary - символьная переменная. Возвращает 'YES', если файл или устройство опознаны как двоичные, и 'NO' или 'UNKNOWN' - в противном случае.

blank - символьная переменная. Возвращает 'NULL', если действует дескриптор преобразования BN, и возвращает 'ZERO', если действует дескриптор BZ. Возвращает 'UNDEFINED' при отсутствии подсоединения или если файл открыт не для форматного В/В.

blocksize - переменная стандартного целого типа. Возвращает размер буфера В/В в байтах. Возвращает 0 при отсутствии подсоединения.

bf - символьная переменная. Возвращает 'YES' ('NO'), если файл или устройство подсоединены и действует (не действует) буферизация, или 'UNKNOWN', если файл или устройство не подсоединены.

carriagecontrol - символьная переменная, возвращающая 'FORTRAN', если первый символ форматной записи трактуется как символ управления кареткой, или 'LIST', если первый символ форматных файлов ничем не отличается от других символов записи.

delim - символьная переменная, возвращающая 'APOSTROPHE', если для символьных данных при управляемом списком В/В в качестве ограничителя используется апостроф ('). Возвращает 'QUOTE', если ограничителями являются кавычки ("). Возвращает 'NONE', если ограничитель не задан. Возвращает 'UNDEFINED' при отсутствии подсоединения.

direct - символьная переменная. Возвращает 'YES', если опрашиваемое устройство или файл открыты для прямого доступа, и возвращает 'NO' или 'UNKNOWN' в противном случае.

err - метка исполняемого оператора. При возникновении ошибки управление передается на оператор, имеющий метку *err*.

exist - логическая переменная; возвращает .TRUE., если опрашиваемое устройство или файл существует, или .FALSE. - в противном случае.

fm - символьная переменная. Возвращает одно из перечисленных в разд. 11.4 для спецификатора CONVERT значений, или 'UNKNOWN', если файл или устройство не подсоединены для неформатной передачи данных.

form - символьная переменная. Возвращает 'FORMATTED', если устройство или файл подсоединены для форматного В/В; возвращает 'UNFORMATTED' при неформатном В/В и возвращает 'BINARY' при двоичном В/В. Возвращает 'UNDEFINED' при отсутствии подсоединения.

formatted - символьная переменная. Возвращает 'YES', если устройство или файл открыты для форматного В/В, и 'NO' - в противном случае. Возвращает 'UNKNOWN', если процессор не может определить, какой В/В разрешен.

iofocus - переменная стандартного логического типа. Возвращает .TRUE., если заданное устройство (окно приложения QuickWin) находится в фокусе, в противном случае возвращает .FALSE.. Параметр может быть использован только с QuickWin-приложениями.

iostat - переменная стандартного целого типа. Возвращает 0 при отсутствии ошибок, отрицательное число, если возникла ситуация "конец файла", или номер возникшей ошибки.

mode - символьная переменная. Возвращает значения *mode* или *action* ('READ', 'WRITE' или 'READWRITE'), заданные для устройства (файла) оператором OPEN. Возвращает 'UNDEFINED' при отсутствии подсоединения.

name - символьная переменная. Возвращает при опросе устройства имя подсоединенного к нему файла. Если файл не подсоединен к устройству или если подсоединенный файл не имеет имени, значение переменной *name* не определено. При опросе файла *name* возвращает заданное имя файла.

named - логическая переменная. Возвращает .TRUE., если файл имеет имя, и .FALSE. - в противном случае.

nextrec - переменная стандартного целого типа. Возвращает номер следующей записи в файле прямого доступа. Номер первой записи файла равен единице.

num - переменная стандартного целого типа. При опросе файла возвращает номер подсоединенного к файлу устройства. Если к файлу не подсоединено устройство, значение переменной *num* не определено. При опросе устройства переменная *num* возвращает номер опрашиваемого устройства. Если задано UNIT = *, то нельзя включать спецификатор NUMBER.

opened - логическая переменная, возвращающая при опросе устройства .TRUE., если какой-либо файл подсоединен к устройству, и .FALSE. - в противном случае. При опросе файла возвращает .TRUE., если файл подсоединен к какому-либо устройству, и .FALSE. - в противном случае.

org - символьная переменная, возвращает значение 'SEQUENTIAL', если файл имеет последовательную организацию, 'RELATIVE', если - связанную, или 'UNKNOWN', если процессор не может определить вид организации файла.

pad - символьная переменная, возвращающая 'YES', если файл открыт с PAD = 'YES', и 'NO' - в противном случае.

position - символьная переменная, возвращающая 'REWIND', если файл позиционирован в своей начальной точке. Возвращает 'APPEND', если файл расположен в своей конечной точке перед записью "конец файла". Возвращает 'ASIS', если файл подсоединен без изменения позиции. Возвращает 'UNDEFINED' при отсутствии подсоединения или если файл подсоединен для прямого доступа.

read - символьная переменная, возвращающая 'YES', если файл открыт для чтения, и 'NO', если из файла нельзя вводить данные. Возвращает 'UNKNOWN', если процессор не может определить, разрешается ли читать из файла.

readwrite - символьная переменная, возвращающая 'YES', если файл открыт как для чтения, так и для записи, и 'NO', если нельзя выполнять

чтение или запись. Возвращает 'UNKNOWN', если процессор не может определить, разрешается ли использовать файл и для чтения и для записи.

recl - переменная стандартного целого типа. Возвращает длину записи (в байтах) файла прямого доступа. Если файл подсоединен для неформатной передачи данных, возвращаемое число байт зависит от используемой операционной системы.

rtyp - символьная переменная. Возвращает одно из перечисленных в разд. 11.4 для спецификатора RECORDTYPE значений, или 'UNKNOWN', если файл или устройство не подсоединены.

seq - символьная переменная. Возвращает 'YES', если файл подсоединен для последовательного доступа, и 'NO' или 'UNKNOWN' - в противном случае.

share - символьная переменная. Возвращает значение статуса *share*, заданного файлу оператором OPEN: 'COMPAT', 'DENYRW', 'DENYWR', 'DENYRD' и 'DENYNONE'. При опросе устройства, если к устройству не подсоединен файл, значение переменной *share* не определено.

unformatted - символьная переменная. Возвращает 'YES', если файл открыт для неформатной передачи данных, и 'NO' - в противном случае. Возвращает 'UNKNOWN', если процессор не может определить, какой В/В разрешен.

write - символьная переменная, возвращающая 'YES', если файл открыт для записи, и 'NO', если в файл нельзя выводить данные. Возвращает 'UNKNOWN', если процессор не может определить, допустимо ли выводить в файл данные.

Оператор INQUIRE возвращает значение атрибутов, с которыми файл был открыт. Свойства неоткрытых файлов не могут быть возвращены оператором. Если некоторые атрибуты не заданы, то оператор возвращает установленные для них по умолчанию значения.

В качестве применяемых в операторе INQUIRE переменных могут быть использованы простые переменные, элементы массивов и компоненты производных типов.

Если при опросе устройства ключевое слово UNIT= опущено, то параметр *u* должен идти первым. Другие параметры могут располагаться в произвольном порядке, но не должны повторяться.

Пример:

```
character(25) :: st(25) = ' '
open(1, file = 'a.txt', action = 'write', position = 'append')
write(1, *) st
rewind 1
inquire(1, name = st(1), action = st(2), blank = st(3), position = st(4))
print *, (trim(st(i)), ' ', i = 1, 4)              ! a.txt WRITE NULL REWIND
```

11.15. Функция EOF

Функция возвращает .TRUE., если подсоединенный к устройству *u*-файл позиционирован на специальной записи "конец файла" или после этой записи. Иначе EOF возвращает .FALSE. Синтаксис функции:

flag = EOF(*u*)

u - устройство В/В (разд. 10.2).

11.16. Организация быстрого ввода/вывода

Затраты времени на В/В снизятся, если придерживаться следующих правил:

1) по возможности используйте неформатные файлы вместо форматных. Так, вывод в файл, подсоединенный к устройству 10, выполнится значительно быстрее вывода в файл, который подсоединен к устройству 20:

```
real(4), dimension(100, 20) :: array = 1.0
open(10, file = 'a.dat', form = 'unformatted')    ! или form = 'binary'
open(20, file = 'a.txt', form = 'formatted')
write(10) array                                   ! Доступ к файлу a.dat происходит
write(20, '(20f8.3)') array                       ! быстрее, чем к файлу a.txt
```

2) выполняйте В/В всего массива или всей строки, не используя циклов;
3) если все же при передаче многомерных массивов необходимо организовать вложенные циклы, то применяйте естественный порядок передачи элементов массива: во внутреннем цикле должен изменяться самый левый индекс, а во внешнем - самый правый. Это обеспечит доступ к элементам массива в порядке их размещения в памяти, что, понятно, ускорит передачу данных;
4) используйте, если позволяют ресурсы, для хранения промежуточных результатов оперативную память, а не внешние файлы;
5) применяйте в случае форматного В/В при программировании формата целочисленное выражение вместо символьной строки, поскольку в первом случае формат определяется единожды - при компиляции, а во втором - спецификация формата, строка *form*, - формируется в процессе исполнения программы:

```
real(4), dimension(1000) :: array
integer(4) :: i
character(15) :: form
...                                               ! Вычисляется значение переменной n
! Этот способ заданий формата лучше, чем формирование строки  form,
! содержащей спецификацию формата
! <n> - выражение в дескрипторе преобразований
print '(1x, <n> f8.3)', (array(i), i = 1, n)
```

```
write(form, '(a, i5, a)') '(1x', n, 'f8.3)'      ! Формируем строку формата form
print form, (array(i), i = 1, n)                 ! Вывод по формату form
```

6) создавайте условия для декомпозиции используемых в операторах В/В циклических списков. Для этого переменная цикла должна быть целочисленной, не должна быть формальным параметром, принадлежать оператору EQUIVALENCE и обладать атрибутом VOLATILE, а спецификация формата в случае форматной передачи данных не должна иметь целочисленных выражений в дескрипторе преобразований. Пример циклического списка:

```
write(10, '(20f8.3)') (array(i), i = 1, n)       ! Циклический список из n элементов
```

Пояснение. Обычно каждый элемент списка В/В обращается к процедурам В/В библиотеки CVF. Временные затраты на эти обращения значительны. С целью их уменьшения неявный цикл замещается компилятором на несколько (до семи) вложенных DO-циклов, использующих для вывода оптимизированную процедуру В/В, которая может осуществлять передачу порциями, содержащими несколько элементов В/В;

7) для увеличения объема передаваемых данных при одном обращении к диску попытайтесь увеличить значение спецификатора BUFFER-COUNT оператора OPEN, но не меняйте значение спецификатора BLOCKSIZE того же оператора, поскольку оно подбирается оптимальным для используемого устройства. Влияние BUFFERCOUNT на скорость передачи данных должно быть установлено экспериментально;
8) не задавайте значение спецификатора RECL оператора OPEN большим, чем размер буфера В/В (этот размер определяется спецификатором BLOCKSIZE), так как передача избыточных данных, незначительно заполняющих буфер, малопроизводительна;
9) значение спецификатора RECL = *recl* выбирайте таким образом, чтобы буфер В/В заполнялся наилучшим образом. Буфер будет заполнен полностью, если его размер кратен *recl* или, наоборот, значение RECL кратно размеру буфера, например: размер буфера равен 8192, а в операторе OPEN задан спецификатор RECL = 1024 или RECL = 16384;
10) используйте оптимальный с позиции быстродействия тип записей, задаваемый в операторе OPEN спецификатором RECORDTYPE:

- для файлов с последовательной организацией наибольшую производительность обеспечит задание записей фиксированной длины (RECORDTYPE = 'FIXED');
- в случае неформатных файлов последовательной организации используйте записи переменной длины (RECORDTYPE = 'VARIABLE');

- в форматных файлах последовательной организации с записями переменной длины задавайте RECORDTYPE = 'STREAM_LF'.

12. Конструктор модулей для объектов ActiveX

12.1. Некоторые сведения об объектах ActiveX

Технология использования в приложениях, в том числе и написанных на Фортране, объектов, созданных в других приложениях, называется *Автоматизацией*, ранее известная как *OLE Автоматизация*. Сами же созданные в рамках этой технологии объекты называются *объектами ActiveX*. Доступ к объекту осуществляется при помощи интерфейса или напрямую через таблицу виртуальных функций. Объекты ActiveX поддерживают модель многокомпонентных объектов COM.

Интерфейс Автоматизации - это группа взаимосвязанных функций, обеспечивающих доступ к методам и свойствам объекта, а также обработку распознаваемых объектом событий. *Метод* - это действие, которое может выполнять объект. *Свойство* - это функция-член, обеспечивающая доступ к информации о состоянии объекта. Большинство свойств имеет две функции доступа - одна возвращает значение свойства, вторая его устанавливает. *Событие* - это действие, распознаваемое объектом, например щелчок мышью или нажатие клавиши. Событие является методом, вызываемым объектом. В общем случае объект может поддерживать несколько интерфейсов. Указатель на любой из них вернет подпрограмма COMQueryInterface.

Приложение, осуществляющее доступ к объектам ActiveX, называется *клиентом ActiveX*. Сам же *объект ActiveX* - это экземпляр класса, предоставляющий клиенту ActiveX свои свойства, методы и события. Объекты ActiveX создаются *компонентами ActiveX*, в качестве которых могут выступать или приложения, или библиотеки. Так, компонентом ActiveX является Microsoft Excel. Как правило, компонент ActiveX предоставляет множество объектов. Например, Excel содержит объект с именем Application (приложение), для инициализации и создания которого в Фортран-приложении потребуется выполнить команды

```
call COMInitialize(status)          ! Инициализируем COM и создаем объект Excel
call COMCreateObject("Excel.Application.8", excelapp, status)
```

Свойство Visible объекта *excelapp* изменит команда

```
call $Application_SetVisible(excelapp, .true.)
```

Кроме чисто объектов, компонент ActiveX может предоставлять *объекты-наборы*, которые в общем случае состоят из различных экземпляров некоторого объекта. Например, Excel может предоставить клиенту ActiveX несколько "Рабочих книг", а в пределах каждой

"Рабочей книги" - несколько "Рабочих листов". Создание объекта-набора "Рабочая книга" обеспечит команда

```
! Получаем указатель на объект "Рабочая книга" - Workbooks
workbooks = $Application_GetWorkbooks(excelapp, $status = status)
```

Экземпляр этого объекта вернет вызов

```
! Открываем заданный файл. Указываем в качестве параметра имя файла Excel
workbook = Workbooks_Open(workbooks, fname, $status = status)
```

12.2. Для чего нужен конструктор модулей

Процедуры, обеспечивающие работу с объектами ActiveX, можно разделить на две группы:

1) процедуры управления объектами. Они применяются со всеми объектами Автоматизации;
2) процедуры, реализующие объекты, их методы и свойства и реагирующие на события. В общем случае подобные процедуры уникальны (как по именам, так и по функциям) для каждого объекта, его метода, свойства или события.

Процедуры первой группы реализованы в поставляемых с CVF библиотеках dfcom.lib и dfauto.lib. Для доступа к ним в CVF имеются необходимые интерфейсы. Они нужны, поскольку эти процедуры написаны на СИ. Применяемая в Фортране технология создания интерфейсов в разноязычных приложениях рассмотрена в [1].

Процедуры второй группы описаны в сопровождающей компоненты ActiveX документации. Поэтому, чтобы ими воспользоваться, нужно иметь под рукой необходимые источники информации. Однако перед программирующем на Фортране пользователем, даже имеющим необходимые сведения, возникают серьезные проблемы, связанные с обеспечением доступа к процедурам. Что опять-таки связано с тем, что они реализованы, как правило, не на Фортране.

Чтобы облегчить доступ к процедурам второй группы, с CVF поставляется *конструктор модулей* Visual Fortran Module Wizard (далее - КМ), создающий по имеющейся об объектах информации модули на Фортране, содержащие описания используемых с объектами данных и тексты процедур второй группы. (Информация о процедурах размещена в соответствующих файлах. Так, все данные об объектах Excel находятся в поставляемой с Excel библиотеке Excel8.olb.)

Далее мы рассмотрим общие для всех объектов ActiveX процедуры управления Автоматизацией и употребляемые в Автоматизации виды данных. А затем - порядок работы с КМ и разберем пример его применения.

12.3. Интерфейсы процедур управления Автоматизацией

Интерфейсы процедур, имена которых приведены в табл. 12.1, содержатся в файле dfcom.f90. Процедуры обеспечивают инициализацию объекта ActiveX, его создание, активизацию и освобождение. Управление свойствами объекта и вызов связанных с ним методов осуществляется процедурами, приведенными в табл. 12.2. Интерфейсы к ним находятся в файле dfauto.f90.

Таблица 12.1. Процедуры, управляющие объектами ActiveX

Процедура	*Назначение*	*Вид*
COMAddObject Reference	Добавляет ссылку на объект	Функция типа INTEGER(4)
COMCLSIDFrom ProgID	Передает программный идентификатор и возвращает соответствующий идентификатор класса	Подпрог-рамма
COMCLSIDFrom String	Передает строку, содержащую идентификатор класса, и возвращает соответствующий идентификатор класса	То же
COMCreateObject ByGUID	Передает идентификатор класса и создает экземпляр объекта. Возвращает указатель на интерфейс объекта	"
COMCreateObject ByProgID	Передает программный идентификатор и создает экземпляр объекта. Возвращает указатель на IDispatch-интерфейс объекта	"
COMGetActiveObject ByGUID	Передает идентификатор класса и возвращает указатель на интерфейс активного в данный момент объекта	"
COMGetActiveObject ByProgID	Передает программный идентификатор и возвращает указатель на IDispatch-интерфейс активного в данный момент объекта	"
COMGetFileObject	Передает имя файла и возвращает указатель на IDispatch-интерфейс объекта Автоматизации, который может обрабатывать файл	"
COMQueryInterface	Передает идентификатор интерфейса и возвращает указатель	"
COMReleaseObject	Освобождает объект	Функция типа INTEGER(4)
COMInitialize	Инициализация COM-библиотеки	Подпрог-рамма

COMUninitialize	Освобождение COM-библиотеки (последняя вызываемая COM-подпрограмма)	То же

Таблица 12.2. Процедуры, обеспечивающие управление свойствами объекта и вызовы его методов

Процедура	*Назначение*	*Вид*
AUTOAddArg	Передает имя параметра и значение и добавляет параметр в структуру данных, содержащую список параметров	Подпрограмма
AUTOAllocate InvokeArgs	Размещает структуру со списком параметров, которые будут переданы родовой функции AUTOInvoke	Функция типа INTEGER(4)
AUTODeallocate InvokeArgs	Освобождает память, занимаемую структурой со списком параметров	Подпрограмма
AUTOGetExceptInfo	Запрашивает данные об исключении, с которым завершился метод	То же
AUTOGetProperty	Передает имя или идентификатор свойства и возвращает значение свойства объекта Автоматизации	Функция типа INTEGER(4)
AUTOGetProperty ByID	Передает ID-имя свойства и возвращает значение свойства объекта Автоматизации	То же
AUTOGetProperty InvokeArgs	Передает структуру со списком параметров и возвращает величину свойства объекта Автоматизации	"
AUTOInvoke	Передает имя идентификатора метода объекта и структуру со списком параметров и вызывает соответствующий метод	"
AUTOSetProperty	Передает имя идентификатора свойства и его величину и устанавливает значение свойства объекта Автоматизации	"
AUTOSetProperty ByID	Передает ID-имя свойства и его величину и устанавливает значение свойства объекта Автоматизации	"
AUTOSetProperty InvokeArgs	Передает структуру со списком параметров и устанавливает заданное значение свойства объекта Автоматизации	"

12.4. Идентификация объекта

Идентификация объекта ActiveX выполняется по глобальному уникальному идентификатору GUID (globally unique identifier), определенному в файле dfwinty.f90:

```
type guid
  sequence
  integer*4 data1
  integer*2 data2
  integer*2 data3
  character*8 data4
end type guid
```

COM использует GUID для идентификации классов, интерфейсов и других требующих уникальных идентификаторов характеристик объекта. Чтобы создать экземпляр объекта, необходимо сообщить COM его (объекта) GUID. Также COM поддерживает программный идентификатор ProgID (programmatic identifier), имеющий вид:

```
application_name.object_name.object_version
```

Например: Excel.Application.8. Тип ProgID - CHARACTER(*). Соответствие между программным идентификатором и идентификатором класса устанавливается подпрограммой COMCLSIDFromProgID.

12.5. Примеры работы с данными Автоматизации

С Автоматизацией связаны специальные виды данных, например разнообразные константы, BSTR-строки, OLE-массивы или варианты, и процедуры, выполняющие с ними определенные действия, например размещение данных в памяти, изменение их значений или преобразование типов. В CVF эти данные определены в файле dfwinty.f90, а интерфейсы связанных с ними процедур - в файле oleaut32.f90. Собственно процедуры реализованы в файле oleaut32.dll.

Для детального ознакомления с видами данных Автоматизации можно рекомендовать, например, приведенную в разд. 12.2 литературу. Здесь же приведем понятия OLE-массивов, BSTR-строк и вариантов, перечислим процедуры, работающие с этими объектами, и дадим ряд примеров. Заметим, что приводимые далее процедуры описаны также и в поставляемом с CVF файлом помощи.

Навыки работы с данными Автоматизации нужны программисту, использующему COM-технологии, в частности, для того, чтобы обеспечить обмен данными между процедурами Фортрана и Автоматизации. При этом потребуется выполнять операции по преобразованию типов, например переходить от строки Фортрана к BSTR-строке, и данных, например передавать данные из массива Фортрана в OLE-массив.

12.5.1. OLE-массивы

Массивы, связанные с диспетчерским интерфейсом IDispatch, называются *OLE-массивами*. Их другое название - *безопасные массивы*. Внутри OLE-массива содержится информация о его ранге и форме. Для доступа к массиву используется его дескриптор, возвращаемый функцией

SafeArrayCreate . Данные массива размещаются в памяти так же, как и данные массива Фортрана: быстрее всего изменяется самый первый индекс. В частности, в случае двумерного OLE-массива его данные размещаются в памяти компьютера по столбцам.

С OLE-массивами связаны приведенные в табл. 12.3 функции. Все они имеют тип INTEGER(4).

Таблица 12.3. Функции, работающие с OLE-массивами

Функция	*Назначение*
SafeArrayAccessData	Увеличивает счетчик блокировок массива и возвращает указатель на данные массива
SafeArrayAllocData	Выделяет память для OLE-массива, используя дескриптор, возвращенный SafeArrayAllocDescriptor
SafeArrayAllocDescriptor	Выделяет память для дескриптора массива
SafeArrayCopy	Копирует массив
SafeArrayCopyData	Копирует исходный массив в другой, предварительно освободив содержимое последнего
SafeArrayCreate	Создает новый дескриптор массива
SafeArrayCreateVector	Создает вектор заданного размера
SafeArrayDestroy	Разрушает дескриптор массива
SafeArrayDestroyData	Освобождает память, занятую массивом
SafeArrayDestroyDescriptor	Освобождает память, занятую дескриптором массива
SafeArrayGetDim	Возвращает ранг массива
SafeArrayGetElement	Возвращает элемент массива
SafeArrayGetElemsize	Возвращает размер элемента массива в байтах
SafeArrayGetLBound	Возвращает нижнюю границу для заданного измерения
SafeArrayGetUBound	Возвращает верхнюю границу для заданного измерения
SafeArrayLock	Увеличивает число блокировок массива
SafeArrayPtrOfIndex	Возвращает указатель на элемент массива
SafeArrayPutElement	Присваивает значение элементу массива
SafeArrayRedim	Изменяет правую, менее значимую границу массива
SafeArrayUnaccessData	Уменьшает счетчик блокировок массива и делает недействительным указатель, возвращенный SafeArrayAccessData
SafeArrayUnlock	Уменьшает счетчик блокировок массива

Пример. Первоначально создается OLE-массив, по форме совпадающий с массивом Фортрана *a*, затем данные из массива *a* переносятся в OLE-массив, после чего его содержимое отображается на экране.

```
program SafeArrayTest
 use dfcomty                          ! Модуль ссылается на DFWINTY
 use dfcom                            ! Модуль ссылается на OLEAUT32 и DFWINTY
 implicit none
 integer(4) :: result, i, j, value
 integer(4) :: a(4, 3)                ! Массив Фортрана
 type(sa_bounds) :: ab(2)             ! Тип sa_bounds описан в модуле DFWINTY
 integer(4) :: indices(2)
 integer(4) :: safeArray              ! OLE-массив
! Задание массива по столбцам
a = reshape((/      11,   12,   13,   14,        &        ! Столбец 1
                    21,   22,   23,   24,        &        ! Столбец 2
                    31,   32,   33,    34 /), shape = (/ 4, 3 /))   ! Столбец 3
! Формируем OLE-массив
ab(1)%lbound = 1                      ! Нижняя граница по первому измерению
ab(1)%extent = ubound(a, 1)           ! Протяженность по первому измерению
ab(2)%lbound = 1
ab(2)%extent = ubound(a, 2)
! Создаем новый дескриптор массива
! Константа VT_I4 описана в файле dfwinty.f90
! Она означает, что OLE-массив содержит 4-байтовые целые числа
safeArray = SafeArrayCreate(VT_I4, 2, ab(1))
! Интерфейсы функций, работающих с OLE-массивами, см. в oleaut32.f90
do j = ab(2)%lbound, ab(2)%extent     ! Переносим данные в OLE-массив
 do i = ab(1)%lbound, ab(1)%extent
  indices(1) = i; indices(2) = j
  ! Переносим данные из массива Фортрана в OLE-массив
  result = SafeArrayPutElement(safeArray, indices(1), loc(a(i, j)))
 end do
end do
do j = ab(2)%lbound, ab(2)%extent     ! Читаем и выводим данные OLE-массива
 do i = ab(1)%lbound, ab(1)%extent
  indices(1) = i; indices(2) = j
  result = SafeArrayGetElement(safeArray, indices(1), loc(value))
  write(*, '(i5)', advance = 'no') value     ! Вывод без продвижения
 end do
 print *                              ! Переход на новую строку
end do
result = SafeArrayDestroy(safeArray)  ! Освобождаем память
end program SafeArrayTest
```

Замечание. В модуле OLEAUT32 интерфейсы определены не для всех функций табл. 12.3. Однако при необходимости недостающий интерфейс можно записать самостоятельно.

Пример. В программе *OLE_vector* создается интерфейс функции SafeArrayCreateVector, которая используется для формирования OLE-вектора. В вектор заносятся элементы строки *string*.

```
program OLE_vector
 use dfwinty                          ! Для получения значения VT_UI1
 use dfcom
 implicit none
 ! Опишем интерфейс SafeArrayCreateVector, поскольку его нет в модуле OLEAUT32
 interface
  integer(4) function SafeArrayCreateVector(vt, lLbound, cElements)
   !dec$ attributes default, stdcall, alias :       &
         '_SafeArrayCreateVector@' :: SafeArrayCreateVector
   !dec$ attributes value :: vt          ! Все параметры передаются по значению
   !dec$ attributes value :: lLbound
   !dec$ attributes value :: cElements
   integer(4), intent(in) :: vt, lLbound , cElements
  end function SafeArrayCreateVector
 end interface
 integer(4) :: safeArray, len, i, indices(1), result
 character(30) :: string = 'Test string'
 character(1) :: ch
 len = len_trim(string)               ! Длина строки string без завершающих пробелов
 ! Создаем новый дескриптор массива
 ! Константа VT_UI1 описана в файле dfwinty.f90
 ! Она означает, что OLE-массив содержит символы
 safeArray = SafeArrayCreateVector(VT_UI1, 1, len)
 do i = 1, len                        ! Заносим данные в OLE-вектор
  indices(1) = i
  result = SafeArrayPutElement(safeArray, indices(1), loc(string(i:i)))
 end do
 do i = 1, len                        ! Контрольный вывод
  indices(1) = i                      ! Читаем и выводим данные из OLE-вектора
   result = SafeArrayGetElement(safeArray, indices(1), loc(ch))
   write(*, '(a)', advance = 'no') ch  ! Вывод без продвижения
 end do
 print *                              ! Переход на новую строку
end program OLE_vector
```

12.5.2. BSTR-строки

Строки, относящиеся к типам и структурам данных интерфейса IDispatch, называются *BSTR-строками*. Эти строки завершаются нулевым символом (*null*) и предваряются целым числом, хранящим их длину. Внутри

строки также могут быть нулевые символы. Фактически BSTR - это указатель на строку. Его тип в Фортране - INTEGER(4).

С BSTR-строками связаны приведенные в табл. 12.4 процедуры. Все они, кроме подпрограммы SysFreeString, являются функциями типа INTEGER(4).

Таблица 12.4. Процедуры для BSTR-строк

Функция	*Назначение*
SysAllocString	Размещает новую строку и копирует в нее строку-параметр
SysAllocStringByteLen	Принимает ANSI-строку и возвращает BSTR, эту строку содержащий
SysAllocStringLen	Размещает новую строку заданной длины и копирует в нее соответствующее число символов из строки-параметра
SysFreeString (подпрограмма)	Освобождает ранее размещенную строку
SysReAllocString	Изменяет размещение строки, копируя в нее передаваемые данные
SysReAllocStringLen	Изменяет размещение строки, копируя в нее заданное число символов
SysStringByteLen	Возвращает длину строки в байтах
SysStringLen	Возвращает длину строки
VectorFromBSTR	Возвращает OLE-вектор, каждый элемент которого равен соответствующему символу BSTR-строки
BSTRFromVector	Возвращает BSTR-строку, каждый символ которой равен соответствующему элементу OLE-вектора

Замечания:

1. Модуль DFCOM, размещенный в файле dfcom.f90, содержит функцию ConvertStringToBSTR, преобразовывающую строку Фортрана в BSTR, и функцию ConvertBSTRToString, выполняющую обратные преобразования.

2. В модуле OLEAUT32 отсутствуют интерфейсы функций SysAllocStringByteLen, VectorFromBSTR и BSTRFromVector.

Пример. Формируется BSTR-строка, содержащая текст Test string. Далее она преобразовывается в строку Фортрана.

```
program BSTR_example
 use dfcom
 integer(4) :: bstr, len
 character(30) :: string = 'Test string', string2 = ' '
 bstr = ConvertStringToBSTR(string)   ! Формируем BSTR-строку с текстом Test string
 ! Преобразовываем BSTR-строку в строку Фортрана
```

```
 len = ConvertBSTRToString(bstr, string2)
 print *, string2                        ! Test string
end program BSTR_example
```

12.5.3. Варианты

Производный тип данных *variant* определен в файле dfwinty.f90:

```
type variant
  sequence
  integer(2) vt
  integer(2) reserved1, reserved2, reserved3
  record /variant_union/ vu
end type variant
```

Структура *variant_union*, использованная при формировании типа *variant*, имеет вид:

```
structure /variant_union/          ! Взята из файла dfwinty.f90
  union
   map
    integer(4) long_val            ! VT_I4
   end map
   map
    character char_val             ! VT_UI1
   end map
   map
    integer(2) short_val           ! VT_I2
   end map
   map
    real(4) float_val              ! VT_R4
   end map
   map
    real(8) double_val             ! VT_R8
   end map
   map
    integer(2) bool_val            ! VT_BOOL
   end map
   map
    integer(4) scode_val           ! VT_ERROR
   end map
   map
    real(8) date_val               ! VT_DATE
   end map
   map                             ! ptr_val - целочисленный указатель
    integer(4) ptr_val             ! Для параметров, передаваемых по ссылке
   end map
  end union
end structure
```

Тип *variant* широко распространен в Автоматизации; им, в частности, обладает подавляющее большинство параметров свойств и методов Автоматизации. Переменные типов *variant* называются *вариантами*. Варианты предназначаются для хранения данных разных типов и позволяют преобразовывать один тип в другой. Возможные типы, используемые с вариантами и другими объектами Автоматизации, задаются определенными в модуле DFWINTY целочисленными константами. Перечислим некоторые из них:

```
integer(2), parameter :: vt_empty = 0       ! Значение не задано
integer(2), parameter :: vt_null = 1        ! null
integer(2), parameter :: vt_i2 = 2          ! INTEGER(2)
integer(2), parameter :: vt_i4 = 3          ! INTEGER(4)
integer(2), parameter :: vt_r4 = 4          ! REAL(4)
integer(2), parameter :: vt_r8 = 5          ! REAL(8)
! Дата - число с плавающей точкой двойной точности
integer(2), parameter :: vt_date = 7
integer(2), parameter :: vt_bstr = 8        ! BSTR
! Указатель на объект, реализующий IDispatch
integer(2), parameter :: vt_dispatch = 9
integer(2), parameter :: vt_error = 10      ! Код ошибки
integer(2), parameter :: vt_bool = 11       ! Истина (#FFFF) или ложь (#0000)
integer(2), parameter :: vt_variant = 12    ! Указатель на вариант
! Указатель на объект, реализующий Iunknown
integer(2), parameter :: vt_unknown = 13
integer(2), parameter :: vt_i1 = 16         ! INTEGER(1)
integer(2), parameter :: vt_ui1 = 17        ! Целое без знака длиной в 1 байт
integer(2), parameter :: vt_ui2 = 18        ! Целое без знака длиной в 2 байта
integer(2), parameter :: vt_ui4 = 19        ! Целое без знака длиной в 4 байта
```

Манипулировать вариантами позволяют приведенные в табл. 12.5 функции и подпрограмма. Тип функций - INTEGER(4).

Таблица 12.5. Процедуры для вариантов

Функция	*Назначение*
VariantChangeType	Преобразовывает вариант в другой тип
VariantChangeTypeEx	Преобразовывает вариант в другой тип, используя идентификатор местности LCID, употребляемый при работе с разноязычными приложениями
VariantClear	Очищает вариант (освобождает память, занимаемую вариантом)
VariantCopy	Копирует вариант
VariantCopyInd	Копирует вариант, выполняя преобразование флага VT_BYREF, обеспечивающего передачу по ссылке, в BYVAL, что гарантирует передачу варианта по

	значению
VariantInit (подпрограмма)	Выполняет инициализацию варианта
VariantTimeToDosDateTime	Преобразовывает время, представленное в виде варианта, в дату и время в формате MSDOS
VariantTimeToSystemTime	Преобразовывает время, представленное в виде варианта, в системное представление времени

Замечание. Модуль OLEAUT32 не содержит интерфейсы функция VariantTimeToDosDateTime и VariantTimeToSystemTime.

Пример. Создаются два варианта. Первый предназначается для хранения указателя BSTR, а второй - для 4-байтового вещественного числа. Каждый вариант получает соответствующее значение: первый - указатель на BSTR-строку "Компак Фортран", второй - число 6.1. Далее второй вариант преобразовывается в тип BSTR; оба варианта переводятся в строки Фортрана и выводится результат.

```
program variant_example
 use dfcomty
 use dfcom
 use TextTransfer                ! Для вывода русского текста в DOS-окно
 integer(4) :: status, length    ! Код модуля TextTransfer см. в прил. 1
 character(80) :: char1, char2
 type(variant) v1, v2            ! Тип variant описан в файле dfwinty.f90
 call VariantInit(v1)            ! Инициализация вариантов
 call VariantInit(v2)
 v1%vt = VT_BSTR                 ! Вариант v1 хранит BSTR-строку
 ! Преобразовываем строку Фортрана в строку BSTR
 ! Преобразовывающую функцию ConvertStringToBSTR см. в файле dfcom.f90
 v1%vu%ptr_val = ConvertStringToBSTR("Компак Фортран")
 v2%vt = VT_R4                   ! 4-байтовое вещественное число
 v2%vu%float_val = 6.1
 ! Интерфейсы функций, работающих с вариантами, см. в oleaut32.f90
 ! Преобразовываем вариант v2 в BSTR-строку
 status = VariantChangeType(v2, v2, 0, VT_BSTR)
 ! Преобразовываем строки BSTR в строки Фортрана
 ! Преобразовывающую функцию ConvertBSTRToString см. в файле dfcom.f90
 length = ConvertBSTRToString(v1%vu%ptr_val, char1)
 length = ConvertBSTRToString(v2%vu%ptr_val, char2)
 print *, trim(RuDosWin(trim(char1) // " " // trim(char2), .false.))
 status = VariantClear(v1)       ! Очищаем вариант
 status = VariantClear(v2)
end program variant_example
```

12.6. Другие источники информации

Технология взаимодействия с помощью COM, включающая в том числе и OLE-автоматизацию, подробно описана, кроме [14], в следующих источниках:

- How OLE and COM Solve the Problems of Component Software Design/by K. Brockschmidt//Microsoft Systems Journal. 1996. Vol. 11, N 5 (May). P. 63-80.
- Inside OLE/Red. by K. Brockschmidt. 2d ed. Redmond; Washington: Microsoft Press, 1995.
- OLE 2 Programmer's Reference, Vol. 2. Redmond; Washington: Microsoft Press, 1994.
- Understanding ActiveX and OLE/Red. by D. Chappell. Redmond; Washington: Microsoft Press, 1996.
- Win 32 SDK, OLE Programmer's Reference online version.
- Win 32 SDK, Automation online version.
- http://mspress.microsoft.com/.

12.7. Как воспользоваться объектом ActiveX

Чтобы использовать объект ActiveX в Фортран-программе, необходимо выполнить следующие действия:

- найти существующий или установить новый объект в системе. Объект можно зарегистрировать специальной программой либо в результате его создания средствами Visual C++ или Visual Basic (см., например, документацию по DS);
- определить вид интерфейса, который объект имеет (в общем случае объект может иметь несколько интерфейсов), и используемые в объекте типы данных. Необходимые об объекте сведения добываются из связанной с ним документации. Также их можно получить, воспользовавшись имеющимся в DS средством просмотра объектов Автоматизации, вызов которого происходит в результате выполнения цепочки Tools - OLE/COM Object Viewer;
- применить КМ и получить код модуля, обеспечивающего доступ к объекту;
- написать программу на Фортране, в которой есть ссылки на полученный модуль и вызовы необходимых для работы с объектом процедур.

12.8. Применение конструктора модулей

Вызов КМ обеспечивает цепочка Tools - Fortran Module Wizard. После ее выполнения в появившемся окне (рис. 12.1) необходимо задать источник, из которого КМ получит данные об объекте.

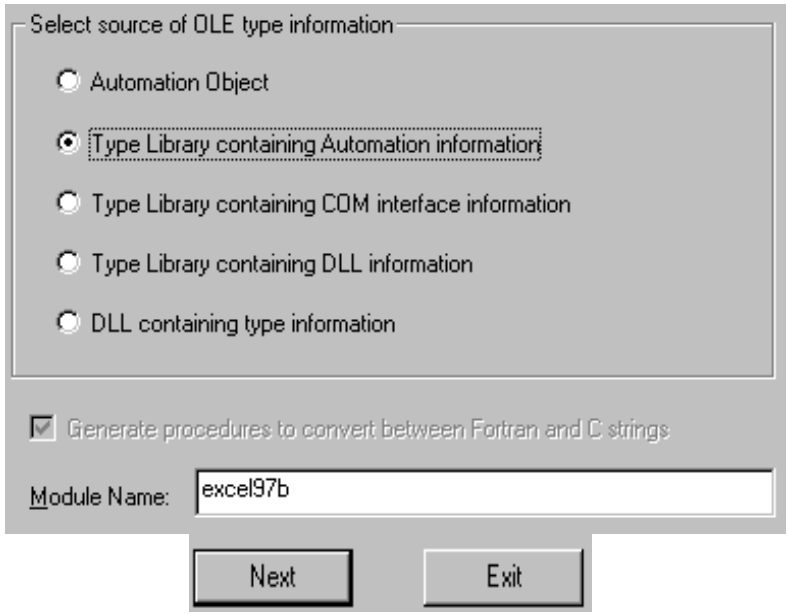


Рис. 12.1. Задание типа объекта

Таким источником может быть:

- сам объект (Automation Object);
- библиотека типа, содержащая данные об объекте Автоматизации (Type Library Containing Automation Information);
- библиотека типа, содержащая данные о COM-интерфейсе объекта (Type Library Containing COM Interface Information);
- библиотека типа, содержащая данные о DLL (Type Library Containing DLL Information);
- библиотека DLL, содержащая данные о типе объекта (DLL Containing Type Information).

Выбор Automation Object производится, когда информация об объекте предоставляется динамически в процессе исполнения приложения. Такая ситуация встречается сравнительно редко, поскольку Microsoft рекомендует, чтобы объекты снабжались библиотекой типа. После выбора Automation Object потребуется ввести имена приложения, объекта и номер версии объекта (рис. 12.2), который, впрочем, может быть опущен. В таком случае будет использована последняя версия, т. е. объект Автоматизации задается в виде application_name.object_name.object_version.

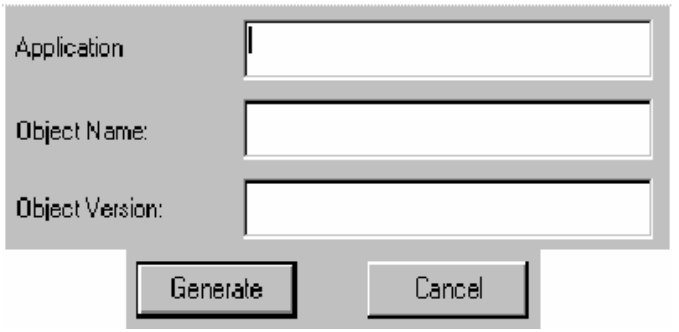


Рис. 12.2. Идентификация объекта

Опция Automation Object может быть использована с объектами, обеспечивающими программный идентификатор (ProgID). Он заносится в системный реестр и идентифицирует исполняемый файл, реализующий объект. Нажатие на кнопку Generate обеспечит формирование модулей, позволяющих использовать объект в Фортран-приложении.

Заданное без расширения в поле Module Name имя (См. рис. 12.1) будет впоследствии использовано для имени формируемого КМ файла. Формируемые КМ файлы имеют расширение F90.

Если выбран другой источник информации о типе, например Type Library Containing Automation Information, то нажатие клавиши Next вызовет появление приведенного на рис. 12.3 экрана.

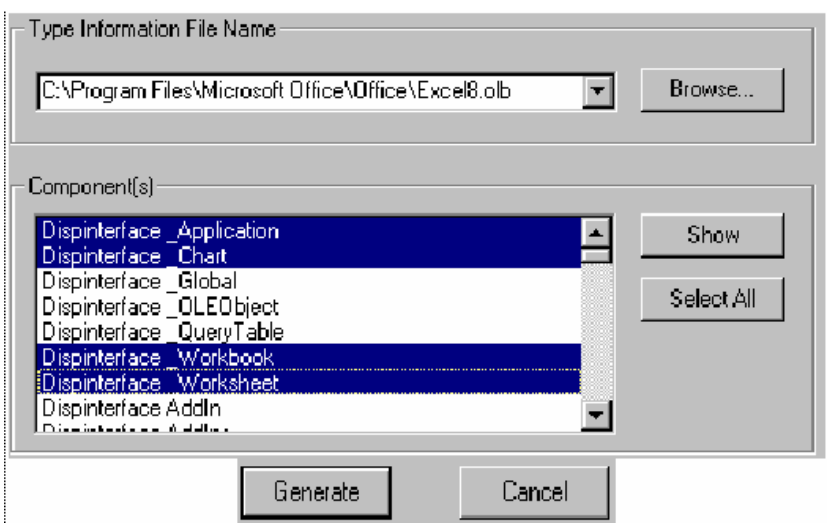


Рис. 12.3. Выбор компонентов из библиотеки типа

Экран позволяет выбрать файл, содержащий библиотеку типа (кнопка Browse), просмотреть состав библиотеки (кнопка Show), выбрать, применив левую кнопку мыши, необходимые или все (кнопка Select All) компоненты. Кнопка Generate обеспечит формирование соответствующих модулей.

Файлы, содержащие библиотеки типов, могут иметь разные расширения, например OLB (объектные библиотеки) или OCX (управляющие элементы ActiveX).

12.9. Пример вызова процедур, сгенерированных конструктором модулей

Сгенерированный КМ файл содержит один или несколько модулей, характеризующих объект и в общем случае включающих:

- определения производных типов данных и констант, обнаруженных в разделе описаний объекта;
- интерфейсы процедур, расположенные в разделе описаний объекта;
- подпрограммы и функции, используемые при работе с объектом.

Имеющиеся в модулях процедуры пригодны для вызова из Фортрана (при наличии соответствующей *use*-ассоциации).

Рассмотрим в качестве примера использования объектов ActiveX написанное на Фортране приложение, выводящее в Excel диаграмму по сформированным в приложении данным.

Проект, создающий приложение, входит в состав поставляемых с CVF образцов и находится в ...\DF98\SAMPLES\ADVANCED\COM\AUTODICE. Состав проекта отображен на рис. 12.4.

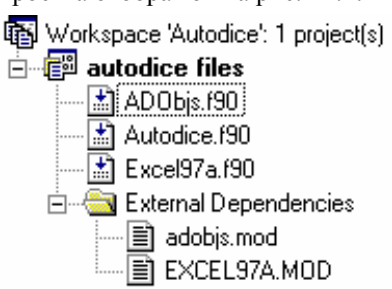


Рис. 12.4. Проект AUTODICE

Главная программа, находящаяся в файле autodice.f90, и модуль ADOBJS написаны программистом. Модуль EXCEL97A получен в результате применения КМ. Для его формирования были выполнены такие действия:

- на приведенном на рис. 12.1 экране выбран источник Type Library Containing Automation Information;
- на следующем экране (рис. 12.3) указан файл c:\Program Files\Microsoft Office\Office\Excel8.olb, содержащий библиотеку с компонентами,

обеспечивающими функционирование Excel, и выбраны компоненты _Application, _Chart. _Workbook, _Worksheet, Axes, Charts, Range, Workbooks, Worksheets, EnumXIAxisGroup, EnaumXIAxisType, необходимые для работы с Excel.

Сгенерированный модуль имеет внушительный объем (около 20'000 строк исходного текста). Ниже приведена его начальная часть, содержащая объявления глобальных констант и одну функцию компонента _Application.

```
! excel97a.f90
! This module contains the Automation interfaces of the objects defined in
! c:\Program Files\Microsoft Office\Office\excel8.olb
! Generated by the Fortran Module Wizard on 10/24/98
module excel97a
 use dfcomty
 use dfauto
 implicit none
 ! CLSIDs
 type(guid), parameter :: CLSID_Global = &
    guid(#00020812, #0000, #0000, &
         char('c0'x)//char('00'x)//char('00'x)//char('00'x)// &
         char('00'x)//char('00'x)//char('00'x)//char('46'x))
 type(guid), parameter :: CLSID_Worksheet = &
   guid(#00020820, #0000, #0000, &
         char('c0'x)//char('00'x)//char('00'x)//char('00'x)// &
         char('00'x)//char('00'x)//char('00'x)//char('46'x))
 type(guid), parameter :: CLSID_Chart = &
    guid(#00020821, #0000, #0000, &
         char('c0'x)//char('00'x)//char('00'x)//char('00'x)// &
         char('00'x)//char('00'x)//char('00'x)//char('46'x))
 type(guid), parameter :: CLSID_APPLICATION = &
    guid(#00024500, #0000, #0000, &
         char('c0'x)//char('00'x)//char('00'x)//char('00'x)// &
         char('00'x)//char('00'x)//char('00'x)//char('46'x))
 ! Enums
 ! XlAxisGroup
 integer, parameter :: xlPrimary = 1
 integer, parameter :: xlSecondary = 2
 ! XlAxisType
 integer, parameter :: xlCategory = 1
 integer, parameter :: xlSeriesAxis = 3
 integer, parameter :: xlValue = 2

 ! Module Procedures
 contains

  function $Application__Evaluate($object, Name, $status)
   !dec$ attributes dllexport :: $Application__Evaluate
   implicit none
```

```
  integer(4), intent(in) :: $object                ! Object Pointer
  !dec$ attributes value :: $object
  type(variant), intent(in) :: Name
  !dec$ attributes reference :: Name
  integer(4), intent(out), optional :: $status     ! Method status
  !dec$ attributes reference :: $status
  integer(4) $$status
  integer(4) invokeargs
  type(variant), volatile :: $return
  type(variant) $Application__Evaluate
  invokeargs = AUTOAllocateInvokeArgs( )
  call AUTOAddArg(invokeargs, '$return', $return, .true.)
  call AUTOAddArg(invokeargs, '$arg1', Name, .false.)
  $$status = AUTOInvoke($object, -5, invokeargs)
  if(present($status)) $status = $$status
  $Application__Evaluate = $return
  call AUTODeallocateInvokeArgs(invokeargs)
 end function $Application__Evaluate
...                                              ! Далее следуют иные процедуры
end module excel97a                              ! модуля
```

Замечания к результатам работы КМ:

1. КМ использует типы данных, имеющиеся в модуле DFCOMTY (а фактически в модуле DFWINTY), и процедуры модуля DFAUTO. Любая процедура сгенерированного модуля EXCEL97A может быть вызвана из создаваемого Фортран-приложения.

2. Если информация о типе содержит комментарий, описывающий функцию-член, то он размещается перед кодом процедуры.

3. Первый параметр сгенерированных процедур всегда имеет имя $*object*. Он является указателем на интерфейс объекта.

4. Директива ATTRIBUTE употребляется для задания соглашения о способах передачи параметров. В частности,

```
!dec$ attributes value :: $object
```

обеспечивает передачу параметра $*object* по значению, а

```
!dec$ attributes reference :: Name
```

говорит о том, что параметр *Name* передается по ссылке.

5. Почти каждая COM-функция-член возвращает статус завершения типа HRESULT, соответствующего типу INTEGER(4).

6. Интерфейс COM-функции-члена подобен интерфейсу DLL-функции. Однако в отличие от последней адрес COM-функции-члена неизвестен построителю приложения. Поэтому для ее вызова необходимо получить

указатель на интерфейс объекта; адрес соответствующей функции-члена вычисляется по значению этого указателя.

Разберем подробнее рассматриваемый пример. Выберем для этого из модулей DFWINTY, DFCOM, OLEAUT32, DFNLS и EXCEL97A код, необходимый для решения поставленной задачи - отображения Фортран-массива в ячейках листа Excel и построения соответствующей диаграммы. Разместим данные, выбранные из модуля DFWINTY, в модуле MYCOMTY, код, взятый из модулей DFCOM, OLEAUT32 и DFNLS, разместим в модуле MYCOM, а код EXCEL97A - в модуле EXCEL97B. Теперь код становится вполне обозримым и пригодным для анализа, выполнить который читателю поможет имеющийся в программе комментарий. В частности, комментарий главной программы включает порядок работы с объектами Excel, придерживаясь которого удается объекты активизировать, задать их свойства и отобразить данные массива *cellCounts* в виде гистограммы.

Для работы приложения необходимо задать имя XLS-файла. В рассмотренном в CVF примере такой файл имеет имя histo.xls и содержит приведенные на рис. 12.5 данные.

	1	2	3	4	5	6	7	8	9	10	11	12
1	66	44	22	1	77	88	99	5	4	33	55	99

Рис. 12.5. Состав файла histo.xls

```
module mycomty                          ! Содержит объявления всех используемых
 !dec$objcomment lib: "dfcom.lib"       ! в приложении autodice данных,
 !dec$objcomment lib: "oleaut32.lib"    ! а также процедур, преобразовывающих
 implicit none                          ! строку Фортрана в BSTR и обратно
 ! Структура variant_union и тип variant взяты из файла dfwinty.f90
 structure /variant_union/
  union
   map
    integer(4) long_val
   end map
   map
    character char_val
   end map
   map
    integer(2) short_val
   end map
   map
    real(4) float_val
   end map
   map
    real(8) double_val
   end map
```

```
   map
    integer(2) bool_val
   end map
   map
    integer(4) scode_val
   end map
   map
    real(8) date_val
   end map
   map
    integer(4) ptr_val                    ! ptr_val - целочисленный указатель
   end map
  end union
 end structure
 type variant
  sequence
  integer(2) vt
  integer(2) reserved1, reserved2, reserved3
  record /variant_union/ vu
 end type variant
 ! Определение типа guid заимствовано из файла dfwinty.f90
 type guid
  sequence
  integer(4) data1
  integer(2) data2, data3
  character(8) data4
 end type guid
 ! Константы vt_i4, vt_bstr, vt_dispatch определены в файле dfwinty.f90
 integer(2), parameter :: vt_i4 = 3, vt_bstr = 8, vt_dispatch = 9
end module mycomty

module mycom                              ! Содержит интерфейсы из модулей DFCOM,
 !dec$objcomment lib: "dfcom.lib"         ! DFAUTO и OLEAUT32 используемых
 !dec$objcomment lib: "dfauto.lib"        ! в приложении autodice процедур,
 !dec$objcomment lib: "oleaut32.lib"      ! а также процедуры, преобразовывающие
 !dec$objcomment lib: "dfnls.lib"         ! строку Фортрана в BSTR и обратно
 implicit none
 ! Родовой интерфейс COMCreateObject взят из модуля DFCOM (файл dfcom.f90)
 interface COMCreateObject
  subroutine COMCreateObjectByProgID(prog_id, idispatch, status)
   !dec$ attributes default :: COMCreateObjectByProgID
   !dec$ attributes reference :: prog_id
   !dec$ attributes reference :: idispatch
   !dec$ attributes reference :: status
   character(*), intent(in) :: prog_id
   integer(4), intent(out) :: idispatch, status
  end subroutine COMCreateObjectByProgID
```

```
 subroutine COMCreateObjectByGUID(clsid, clsctx, iid, iinterface, status)
  use mycomty
  !dec$ attributes default :: COMCreateObjectByGUID
  !dec$ attributes reference :: clsid
  !dec$ attributes reference :: clsctx
  !dec$ attributes reference :: iid
  !dec$ attributes reference :: iinterface
  !dec$ attributes reference :: status
  type(guid), intent(in) :: clsid, clsctx, iid
  integer(4), intent(out) :: iinterface, status
 end subroutine COMCreateObjectByGUID
end interface COMCreateObject

interface

 ! Интерфейсы COMInitialize и COMUninitialize взяты из модуля DFCOM
 subroutine COMInitialize(status)
  !dec$ attributes default :: COMInitialize
  !dec$ attributes reference :: status
  integer(4), intent(out) :: status
 end subroutine COMInitialize

 subroutine COMUninitialize( )
  !dec$ attributes default :: COMUninitialize
 end subroutine COMUninitialize

 ! Интерфейс COMReleaseObject заимствован из модуля DFCOM (файл dfcom.f90)
 integer(4) function COMReleaseObject(iunknown)
  !dec$ attributes default :: COMReleaseObject
  !dec$ attributes value :: iunknown
  integer(4), intent(in) :: iunknown
 end function COMReleaseObject

 ! Интерфейсы функций автоматизации взяты из файла dfauto.f90
 ! Метод активизации параметра
 integer(4) function AUTOAllocateInvokeArgs( )
  !dec$ attributes default :: AUTOAllocateInvokeArgs
 end function AUTOAllocateInvokeArgs
 integer(4) function AUTOSetPropertyByID(idispatch, memid, invoke_args)
  !dec$ attributes default :: AUTOSetPropertyByID
  !dec$ attributes value :: idispatch
  !dec$ attributes value :: memid
  !dec$ attributes value :: invoke_args
  integer(4), intent(in) :: idispatch, memid, invoke_args
 end function AUTOSetPropertyByID

 integer(4) function AUTOGetPropertyByID(idispatch, memid, invoke_args)
  !dec$ attributes default :: AUTOGetPropertyByID
  !dec$ attributes value :: idispatch
  !dec$ attributes value :: memid
  !dec$ attributes value :: invoke_args
  integer(4), intent(in) :: idispatch, memid, invoke_args
```

```
 end function AUTOGetPropertyByID
end interface

! Часть родового интерфейса AUTOSetProperty
interface AUTOSetProperty
 integer(4) function AUTOSetPropertyInteger2Array(idispatch, name, value, type)
  !dec$ attributes default :: AUTOSetPropertyInteger2Array
  !dec$ attributes value :: idispatch
  !dec$ attributes reference :: name
  !dec$ attributes reference :: value
  !dec$ attributes reference :: type
  integer(4), intent(in) :: idispatch
  character(*), intent(in) :: name
  integer(2), dimension(:), intent(in) :: value
  integer(2), intent(in), optional :: type
 end function AUTOSetPropertyInteger2Array

 integer(4) function AUTOSetPropertyInteger4(idispatch, name, value, type)
  !dec$ attributes default :: AUTOSetPropertyInteger4
  !dec$ attributes value :: idispatch
  !dec$ attributes reference :: name
  !dec$ attributes reference :: value
  !dec$ attributes reference :: type
  integer(4), intent(in) :: idispatch, value
  character(*), intent(in) :: name
  integer(2), intent(in), optional :: type
 end function AUTOSetPropertyInteger4
end interface AUTOSetProperty

! Родовой интерфейс AUTOInvoke
interface AUTOInvoke
 integer(4) function AUTOInvokeByName(idispatch, name, invoke_args)
  !dec$ attributes default :: AUTOInvokeByName
  !dec$ attributes value :: idispatch
  !dec$ attributes value :: invoke_args
  !dec$ attributes reference :: name
  integer(4), intent(in) :: idispatch, invoke_args
  character(*), intent(in) :: name
 end function AUTOInvokeByName
```

! ***Замечание.*** При использовании AUTOInvokeByID для всех вызовов
! AUTOAddArg задается параметр "$ARG*nn*"

```
 integer(4) function AUTOInvokeByID(idispatch, memid, invoke_args)
  !dec$ attributes default :: AUTOInvokeByID
  !dec$ attributes value :: idispatch
  !dec$ attributes value :: memid
  !dec$ attributes value :: invoke_args
  integer(4), intent(in) :: idispatch, memid, invoke_args
```

```
 end function AUTOInvokeByID
end interface AUTOInvoke

! Часть родового интерфейса AUTOAddArg
interface AUTOAddArg
 subroutine AUTOAddArgInteger4(invoke_args, name, value, output_arg, type)
  !dec$ attributes default :: AUTOAddArgInteger4
  !dec$ attributes value :: invoke_args
  !dec$ attributes reference :: name
  !dec$ attributes reference :: value
  !dec$ attributes reference :: output_arg
  !dec$ attributes reference :: type
  integer(4), intent(in) :: invoke_args, value
  character(*), intent(in) :: name
  logical(4), intent(in), optional :: output_arg
  integer(2), intent(in), optional :: type
 end subroutine AUTOAddArgInteger4
 subroutine AUTOAddArgLogical2(invoke_args, name, value, output_arg, type)
  !dec$ attributes default :: AUTOAddArgLogical2
  integer(4), intent(in) :: invoke_args
  !dec$ attributes value :: invoke_args
  !dec$ attributes reference :: name
  !dec$ attributes reference :: value
  !dec$ attributes reference :: output_arg
  !dec$ attributes reference :: type
  character(*), intent(in) :: name
  logical(2), intent(in) :: value
  logical(4), intent(in), optional :: output_arg
  integer(2), intent(in), optional :: type
 end subroutine AUTOAddArgLogical2
 subroutine AUTOAddArgCharacter(invoke_args, name, value, output_arg, type)
 !dec$ attributes default :: AUTOAddArgCharacter
  !dec$ attributes value :: invoke_args
  !dec$ attributes reference :: name
  !dec$ attributes reference :: value
  !dec$ attributes reference :: output_arg
  !dec$ attributes reference :: type
  integer(4), intent(in) :: invoke_args
  character(*), intent(in) :: name, value
  logical(4),   intent(in), optional :: output_arg
  integer(2), intent(in), optional :: type
 end subroutine AUTOAddArgCharacter
 subroutine AUTOAddArgVariant(invoke_args, name, value, output_arg)
  !dec$ attributes default :: AUTOAddArgVariant
  !dec$ attributes value :: invoke_args
  !dec$ attributes reference :: name
  !dec$ attributes reference :: value
```

```
  !dec$ attributes reference :: output_arg
  use mycomty
  integer(4), intent(in) :: invoke_args
  character(*), intent(in) :: name
  type(variant), intent(in) :: value
  logical, intent(in), optional :: output_arg
 end subroutine AUTOAddArgVariant
end interface AUTOAddArg

! Интерфейсы SysAllocString, SysStringLen и SysFreeString
! взяты из файла oleaut32.f90
interface
 integer(4) function SysAllocString(unistr)
  !dec$ attributes default, stdcall, alias : '_SysAllocString@' :: SysAllocString
  integer(2), intent(in) :: unistr(*)
 end function SysAllocString

 integer(4) function SysStringLen(bstr)
  !dec$ attributes default, stdcall, alias : '_SysStringLen@' :: SysStringLen
  !dec$ attributes value :: bstr
  integer(4), intent(in) :: bstr
 end function SysStringLen

 subroutine SysFreeString(bstr)
  !dec$ attributes default, stdcall, alias : '_SysFreeString@' :: SysFreeString
  !dec$ attributes value :: bstr
  integer(4), intent(in) :: bstr
 end subroutine SysFreeString

 ! Интерфейсы VariantInit и VariantClear взяты из файла oleaut32.f90
 subroutine VariantInit(pvarg)
  !dec$ attributes default, stdcall, alias : '_VariantInit@' :: VariantInit
  !dec$ attributes reference :: pvarg
  use mycomty                    ! Взамен use dfwinty
  type(variant), intent(out) :: pvarg
 end subroutine VariantInit

 integer(4) function VariantClear(pvarg)
  !dec$ attributes default, stdcall, alias : '_VariantClear@' :: VariantClear
  !dec$ attributes reference :: pvarg
  use mycomty                    ! Взамен use dfwinty
  type(variant), intent(out) :: pvarg
 end function VariantClear
end interface

! Интерфейс функций MBConvertMBToUnicode и MBConvertMBToUnicode взят
! из модуля DFNLS (файл dfnls.f90). Они нужны для работы модульных
! функций ConvertStringToBSTR и ConvertBSTRToString
interface
 integer(4) function MBConvertMBToUnicode(mbstr, unicodestr, flags)
  !dec$ attributes default :: MBConvertMBToUnicode
  character(*), intent(in) :: mbstr
```

```
  integer(2), dimension(:), intent(out) :: unicodestr
  integer(4), intent(in), optional :: flags
 end function MBConvertMBToUnicode
 integer(4) function MBConvertUnicodeToMB(unicodestr, mbstr, flags)
  !dec$ attributes default :: MBConvertUnicodeToMB
  integer(2), dimension(:), intent(in) :: unicodestr
  character(*), intent(out) :: mbstr
  integer(4), optional, intent(in) :: flags
 end function MBConvertUnicodeToMB
end interface

contains

! Процедуры преобразования строки Фортрана в строку BSTR и обратно;
! заимствованы из файла dfcom.f90
integer(4) function ConvertStringToBSTR(string)
 character(*), intent(in) :: string
 integer(4) bstr, length
 integer(2), allocatable :: unistr(:)              ! Строке UNICODE
 ! Первый вызов MBConvertMBToUnicode определяет длину строки string
 allocate(unistr(0))
 length = MBConvertMBToUnicode(string, unistr)
 deallocate(unistr)
 if(length < 0) then                 ! Специальный случай всех пробелов
  allocate(unistr(2))
  unistr(1) = #20                    ! Один пробел
  unistr(2) = 0                      ! Нуль-символ
 else
  ! Второй вызов MBConvertMBToUnicode выполняет преобразование
  allocate(unistr(length + 1))
  length = MBConvertMBToUnicode(string, unistr)
  unistr(length + 1) = 0             ! Завершаем строку нуль-символом
 end if
 bstr = SysAllocString(unistr)    ! Размещаем BSTR-строку
 deallocate(unistr)
 ConvertStringToBSTR = bstr   ! Возвращаем результат
end function ConvertStringToBSTR

integer(4) function ConvertBSTRToString(bstr, string)
 integer(4), intent(in) :: bstr
 character(*), intent(out) :: string
 integer(4) length
 length = SysStringLen(bstr)
 ConvertBSTRToString = Convert(bstr, length, string)

 contains

  integer(4) function Convert(bstr, length, string)
   integer(4), intent(in) :: bstr, length
   character(*), intent(out) :: string
   integer(2) :: unistr(length)
```

```
    pointer(p, unistr)
    p = bstr
    Convert = MBConvertUnicodeToMB(unistr, string)
   end function Convert
  end function ConvertBSTRToString
end module mycom

module adobjs
 implicit none
 ! Указатели на объекты
 integer(4) :: excelapp, workbooks, workbook, worksheets, worksheet, range, charts, chart
 integer(4) :: cells(12)
 integer(4) :: categoryAxis, valueAxis
 integer(4) :: bstr1, bstr2, bstr3

 contains

  subroutine initobjects( )          ! Задает начальные значения переменных
   integer(4) i
   excelapp = 0; workbooks = 0; workbook = 0; worksheets = 0; worksheet = 0
   range = 0; charts = 0; chart = 0; categoryAxis = 0; valueAxis = 0; cells = 0
   bstr1 = 0; bstr2 = 0; bstr3 = 0
  end subroutine initobjects

  subroutine releaseobjects( )       ! Освобождает созданные объекты
   use mycom                         ! Вместо use dfcom
   integer(4) status, i
   if(range /= 0) status = COMReleaseObject(range)
   if(chart /= 0) status = COMReleaseObject(chart)
   if(charts /= 0) status = COMReleaseObject(charts)
   if(worksheets /= 0) status = COMReleaseObject(worksheet)
   if(worksheet /= 0) status = COMReleaseObject(worksheet)
   if(workbook /= 0) status = COMReleaseObject(workbook)
   if(workbooks /= 0) status = COMReleaseObject(workbooks)
   do i =1, 12
    if(cells(i) /= 0) status = COMReleaseObject(cells(i))
   end do
   if(categoryAxis /= 0) status = COMReleaseObject(categoryAxis)
   if(valueAxis /= 0) status = COMReleaseObject(valueAxis)
   if(excelapp /= 0) status = COMReleaseObject(excelapp)
   if(bstr1 /= 0) call SysFreeString(bstr1)
   if(bstr2 /= 0) call SysFreeString(bstr2)
   if(bstr3 /= 0) call SysFreeString(bstr3)
  end subroutine releaseobjects
end module adobjs

program ExcelSample
 ! Взамен ссылок на модули DFCOM, DFCOMTY и EXCEL97B
 use mycom
 use adobjs
 use excel97b
```

```
implicit none
integer(4) status, loopCount, roll, maxScale, i, die(2)
character(32) :: fname
real(4) rnd(2)
integer(2) :: cellCounts(12)              ! Массив, отображаемый в виде диаграммы
type(variant) :: vbstr1, vbstr2, vbstr3, vint
print *, 'Enter Excel file name'
read *, fname
call initobjects( )                       ! Инициализация объектов
cellCounts = 0                            ! Инициализация массива
call COMInitialize(status)                ! Инициализируем COM и создаем объект Excel
call COMCreateObject("Excel.Application.8", excelapp, status)
if(excelapp == 0) stop 'Unable to create Excel object; Aborting'
call $Application_SetVisible(excelapp, .true.)
! Последовательность операций:
! получить указатель на объект "Рабочая книга";
! открыть файл fname и создать экземпляр объекта "Рабочая книга";
! получить указатель на объект "Рабочий лист";
! задать диапазон заполняемых ячеек таблицы;
! заполнить ячейки из выбранного диапазона значениями массива cellCounts;
! задать отображаемый на диаграмме диапазон ячеек таблицы;
! получить указатель на объект "Диаграмма" и сформировать этот объект;
! задать параметры диаграммы и вызвать построитель диаграмм;
! задать параметры осей диаграммы;
! задать максимальную координату на оси значений
! сформировать отображаемый массив cellCounts;
! Передать данные в Excel и отобразить их на диаграмме
! Псевдокод:
! workbooks = excelapp.GetWorkbooks( )
! workbook = workbooks.Open(spreadsheet)
! worksheet = workbook.GetActiveSheet
! range = worksheet.GetRange("A1", "L1")
! range.Select( )
! charts = workbook.GetCharts( )
! chart = charts.Add( )
```

```
! chart.ChartWizard(gallery=chartType, title=title, categoryTitle=title, valueTitle=title)
! valueAxis = chart.Axes(type = xlValue, axisGroup = xlPrimary)
! valueAxis.MaximumScale(loopcount / 5)
! Аналогичный код на Фортране:
! Получаем указатель на объект-набор"Рабочая книга" - workbooks
workbooks = $Application_GetWorkbooks(excelapp, $status = status)
call Check_Status(status, "Unable to get workbooks object")
! Создаем workbook - экземпляр объекта workbooks
! Открываем заданный файл. Указываем в качестве параметра имя XLS-файла
workbook = Workbooks_Open(workbooks, fname, $status = status)
call Check_Status(status, "Unable to get Workbook object; see if the file path is correct")
! Получаем worksheet - указатель на объект "Рабочий лист"
worksheet = $Workbook_GetActiveSheet(workbook, status)
call Check_Status(status, "Unable to get Worksheet object")
call VariantInit(vbstr1)                    ! Создаем новую диаграмму
call VariantInit(vbstr2)
vbstr1%vt = vt_bstr; bstr1 = ConvertStringToBSTR("A1"); vbstr1%vu%ptr_val = bstr1
vbstr2%vt = vt_bstr; bstr2 = ConvertStringToBSTR("L1"); vbstr2%vu%ptr_val = bstr2
! Задаем диапазон заполняемых ячеек таблицы Excel - от A1 до L1
range = $Worksheet_GetRange(worksheet, vbstr1, vbstr2, status)
call Check_Status(status, "Unable to get range object")
status = VariantClear(vbstr1); bstr1 = 0
status = VariantClear(vbstr2); bstr2 = 0
! Заполняем ячейки из выбранного диапазона значениями массива cellCounts
status = AUTOSetProperty(range, "Value", cellCounts)
! Выбираем отображаемый на диаграмме диапазон ячеек
call Range_Select(range, status)
! Получаем указатель на объект "Диаграмма"
charts = $Workbook_GetCharts(workbook, $status = status)
call Check_Status(status, " Unable to get charts object")
chart = Charts_Add(charts, $status = status)
call Check_Status(status, " Unable to add chart object")
! Вызываем построитель диаграмм. Псевдокод:
! chart.ChartWizard(gallery=chartType, title=title, categoryTitle=title, valueTitle=title)
call VariantInit(vint)                      ! Код Фортрана
! Вид гистограммы - объемные вертикальные столбцы
vint%vt = vt_i4; vint%vu%long_val = 11
call VariantInit(vbstr1)                    ! Инициализация варианта
vbstr1%vt = vt_bstr                         ! Тип хранимого значения
```

```
bstr1 = ConvertStringToBSTR("Гистограмма cellCounts"); vbstr1%vu%ptr_val = bstr1
 call VariantInit(vbstr2); vbstr2%vt = vt_bstr
 bstr2 = ConvertStringToBSTR("Столбец"); vbstr2%vu%ptr_val = bstr2
 call VariantInit(vbstr3); vbstr3%vt = vt_bstr
 bstr3 = ConvertStringToBSTR("Значение"); vbstr3%vu%ptr_val = bstr3
 call $Chart_ChartWizard(chart,          &
          Gallery = vint,                &    ! Вид диаграммы
          Title = vbstr1,                &    ! Заголовок диаграммы
          CategoryTitle = vbstr2,        &    ! Заголовок горизонтальной оси
          ValueTitle = vbstr3,           &    ! Заголовок вертикальной оси
          $status = status)
 call Check_Status(status, "Unable to invoke ChartWizard")
 status = VariantClear(vbstr1); bstr1 = 0      ! Очищаем варианты
 status = VariantClear(vbstr2); bstr2 = 0
 status = VariantClear(vbstr3); bstr3 = 0
 call VariantInit(vint)                        ! Устанавливаем свойства осей диаграммы
 vint%vt = vt_i4; vint%vu%long_val = xlValue
 valueAxis = $Chart_Axes(chart, vint, xlPrimary, $status = status)
 call Check_Status(status, "Unable to get axis object")
 loopcount = 1000                              ! Число вызовов датчика случайных чисел
 maxScale = loopcount / 5                      ! Максимальная величина на оси значений
 status = AUTOSetProperty(valueAxis, "MaximumScale", maxScale)
 call Check_Status(status, "Unable to set axis MaximumScale")
 call random_seed( )                           ! Затравка датчика случайных чисел
 do i = 1, loopcount                           ! Формируем отображаемый массив
  call random_number(rnd)                      ! Генерируем два случайных числа
  die = nint((rnd * 6) + 0.5)
  roll = sum(die)
  cellCounts(roll) = cellCounts(roll) + 1
 end do
 ! Отображаем данные массива cellCounts в таблице Excel и на диаграмме
 status = AUTOSetProperty(range, "Value", cellCounts)
 call Check_Status(status, "Unable to set range value")
 call releaseobjects( )                        ! Освобождаем объекты
 call COMUninitialize( )
end program ExcelSample

subroutine Check_Status(olestatus, errorMsg)
 use adobjs
 integer(4) :: olestatus
 character(*) :: errorMsg
 if(olestatus >= 0) return
 call releaseobjects( )                        ! Освобождаем объекты
 write(*, '(a, "; OLE error status = 0x", z8.8, "; Aborting")') trim(errorMsg), olestatus
 stop
end subroutine Check_Status                    ! Результат приведен на рис. 12.6
```

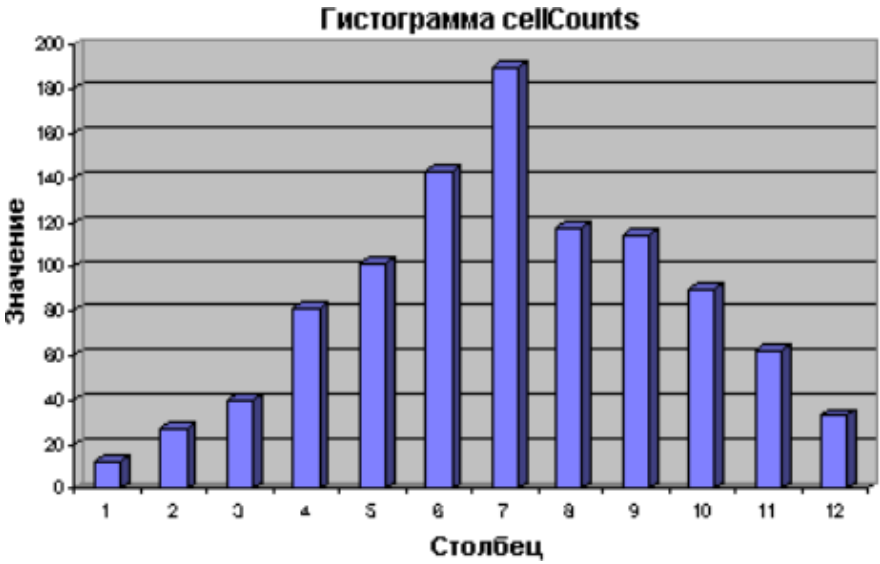


Рис. 12.6. Отображение массива cellCounts на диаграмме Excel

Замечания:

1. Если после запуска приложения были сохранены изменения в файле histo.xls, то его нужно будет восстановить в исходном виде, например списав с CD, содержащем поставку CVF.

2. Все вызываемые из программы ExcelSample процедуры инициализации и задания свойств объектов сосредоточены в модуле EXCEL97B, который сформирован из полученного при помощи КМ модуля EXEL97A.

```
module excel97b
 use mycomty
 use mycom
 implicit none

 ! Объявления констант взяты из файла excel97a.f90
 type(guid), parameter :: CLSID_Global =                    &
      guid(#00020812, #0000, #0000,                         &
           char('c0'x)//char('00'x)//char('00'x)//char('00'x)//  &
           char('00'x)//char('00'x)//char('00'x)//char('46'x))
 type(guid), parameter :: CLSID_Worksheet =                 &
      guid(#00020820, #0000, #0000,                         &
           char('c0'x)//char('00'x)//char('00'x)//char('00'x)//  &
           char('00'x)//char('00'x)//char('00'x)//char('46'x))
 type(guid), parameter :: CLSID_Chart =                     &
      guid(#00020821, #0000, #0000,                         &
```

```
          char('c0'x)//char('00'x)//char('00'x)//char('00'x)//          &
          char('00'x)//char('00'x)//char('00'x)//char('46'x))
type(guid), parameter :: CLSID_Application =                            &
     guid(#00024500, #0000, #0000,                                      &
          char('c0'x)//char('00'x)//char('00'x)//char('00'x)//          &
          char('00'x)//char('00'x)//char('00'x)//char('46'x))
! XlAxisGroup
integer, parameter :: xlPrimary = 1, xlSecondary = 2
! XlAxisType
integer, parameter :: xlCategory = 1, xlSeriesAxis = 3, xlValue = 2
contains                                        ! Модульные процедуры
 ! Все процедуры взяты из файла excel97a.f90
 subroutine $Application_SetVisible($object, $arg1, $status)
  !dec$ attributes dllexport :: $Application_SetVisible
  !dec$ attributes value :: $object
  !dec$ attributes reference :: $arg1
  !dec$ attributes reference :: $status
  implicit none
  integer(4), intent(in) :: $object                 ! Указатель на объект
  logical(2), intent(in) :: $arg1
  integer(4), intent(out), optional :: $status      ! Статус метода
  integer(4) $$status, invokeargs
  invokeargs = AUTOAllocateInvokeArgs( )
  call AUTOAddArg(invokeargs, 'Visible', $arg1)
  $$status = AUTOSetPropertyByID($object, 558, invokeargs)
  if(present($status)) $status = $$status
  call AUTODeallocateInvokeArgs(invokeargs)
 end subroutine $Application_SetVisible
 ! $Application_GetWorkbooks возвращает значение типа POINTER(p, INTEGER(4))
 integer(4) function $Application_GetWorkbooks($object, $status)
  !dec$ attributes dllexport :: $Application_GetWorkbooks
  !dec$ attributes value :: $object
  !dec$ attributes reference :: $status
  implicit none
  integer(4), intent(in) :: $object                 ! Указатель на объект
  integer(4), intent(out), optional :: $status      ! Статус метода
  integer(4) $$status, invokeargs
  integer(4), volatile :: $return
  invokeargs = AUTOAllocateInvokeArgs( )
  call AUTOAddArg(invokeargs, 'Workbooks', $return, .true., vt_dispatch)
  $$status = AUTOGetPropertyByID($object, 572, invokeargs)
  if(present($status)) $status = $$status
  $Application_GetWorkbooks = $return
  call AUTODeallocateInvokeArgs(invokeargs)
 end function $Application_GetWorkbooks
```

```
! Workbooks_Open возвращает значение типа POINTER(p, INTEGER(4))
integer(4) function Workbooks_Open($object, Filename, UpdateLinks, ReadOnly,      &
    Format, Password, WriteResPassword, IgnoreReadOnlyRecommended,                 &
    Origin, Delimiter, Editable, Notify, Converter, AddToMru, $status)
!dec$ attributes dllexport :: Workbooks_Open
!dec$ attributes value :: $object
!dec$ attributes reference :: Filename
!dec$ attributes reference :: UpdateLinks
!dec$ attributes reference :: ReadOnly
!dec$ attributes reference :: Format
!dec$ attributes reference :: Password
!dec$ attributes reference :: WriteResPassword
!dec$ attributes reference :: IgnoreReadOnlyRecommended
!dec$ attributes reference :: Origin
!dec$ attributes reference :: Delimiter
!dec$ attributes reference :: Editable
!dec$ attributes reference :: Notify
!dec$ attributes reference :: Converter
!dec$ attributes reference :: AddToMru
!dec$ attributes reference :: $status
implicit none
integer(4), intent(in) :: $object                  ! Указатель на объект
character(*), intent(in) :: Filename               ! Имя XLS-файла
type(variant), intent(in), optional :: UpdateLinks, ReadOnly, Format, Password,    &
    WriteResPassword, IgnoreReadOnlyRecommended, Origin, Delimiter,                &
    Editable, Notify, Converter, AddToMru
integer(4), intent(out), optional :: $status       ! Статус метода
integer(4) $$status, invokeargs
integer(4), volatile :: $return
invokeargs = AUTOAllocateInvokeArgs( )
! Константы '$RETURN', '$ARGnn' записываются прописными буквами
call AUTOAddArg(invokeargs, '$RETURN', $return, .true., vt_dispatch)
call AUTOAddArg(invokeargs, '$ARG1', Filename, .false., vt_bstr)
if(present(UpdateLinks)) call AUTOAddArg(invokeargs, '$ARG2',                      &
    UpdateLinks, .false.)
if(present(ReadOnly)) call AUTOAddArg(invokeargs, '$ARG3', ReadOnly, .false.)
if(present(Format)) call AUTOAddArg(invokeargs, '$ARG4', Format, .false.)
if(present(Password)) call AUTOAddArg(invokeargs, '$ARG5', Password, .false.)
if(present(WriteResPassword)) call AUTOAddArg(invokeargs, '$ARG6',                 &
    WriteResPassword, .false.)
if(present(IgnoreReadOnlyRecommended)) call AUTOAddArg(invokeargs, '$ARG7', &
    IgnoreReadOnlyRecommended, .false.)
if(present(Origin)) call AUTOAddArg(invokeargs, '$ARG8', Origin, .false.)
if(present(Delimiter)) call AUTOAddArg(invokeargs, '$ARG9', Delimiter, .false.)
if(present(Editable)) call AUTOAddArg(invokeargs, '$ARG10', Editable, .false.)
if(present(Notify)) call AUTOAddArg(invokeargs, '$ARG11', Notify, .false.)
if(present(Converter)) call AUTOAddArg(invokeargs, '$ARG12', Converter, .false.)
```

```
 if(present(AddToMru)) call AUTOAddArg(invokeargs, '$ARG13', AddToMru, .false.)
 $$status = AUTOinvoke($object, 682, invokeargs)
 if(present($status)) $status = $$status
 Workbooks_Open = $return
 call AUTODeallocateInvokeArgs(invokeargs)
end function Workbooks_Open

integer(4) function $Workbook_GetActiveSheet($object, $status)
 !dec$ attributes dllexport :: $Workbook_GetActiveSheet
 !dec$ attributes value :: $object
 !dec$ attributes reference :: $status
 implicit none
 integer(4), intent(in) :: $object                ! Указатель на объект
 integer(4), intent(out), optional :: $status     ! Статус метода
 integer(4) $$status, invokeargs
 integer(4), volatile :: $return
 invokeargs = AUTOAllocateInvokeArgs( )
 call AUTOAddArg(invokeargs, 'ActiveSheet', $return, .true., vt_dispatch)
 $$status = AUTOGetPropertyByID($object, 307, invokeargs)
 if(present($status)) $status = $$status
 $Workbook_GetActiveSheet = $return
 call AUTODeallocateInvokeArgs(invokeargs)
end function $Workbook_GetActiveSheet

! $Worksheet_GetRange возвращает переменную типа POINTER(p, INTEGER(4))
integer(4) function $Worksheet_GetRange($object, Cell1, Cell2, $status)
 !dec$ attributes dllexport :: $Worksheet_GetRange
 !dec$ attributes value :: $object
 !dec$ attributes reference :: Cell1
 !dec$ attributes reference :: Cell2
 !dec$ attributes reference :: $status
 implicit none
 integer(4), intent(in) :: $object                ! Указатель на объект
 type(variant), intent(in) :: Cell1
 type(variant), intent(in), optional :: Cell2
 integer(4), intent(out), optional :: $status     ! Статус метода
 integer(4) $$status, invokeargs
 integer(4), volatile :: $return
 invokeargs = AUTOAllocateInvokeArgs( )
 ! Первая буква константы 'Range' - прописная
 call AUTOAddArg(invokeargs, 'Range', $return, .true., vt_dispatch)
 ! Константы '$ARG1', '$ARG2' записываются прописными буквами
 call AUTOAddArg(invokeargs, '$ARG1', Cell1, .false.)
 if(present(Cell2)) call AUTOAddArg(invokeargs, '$ARG2', Cell2, .false.)
 $$status = AUTOGetPropertyByID($object, 197, invokeargs)
 if(present($status)) $status = $$status
 $Worksheet_GetRange = $return
 call AUTODeallocateInvokeArgs(invokeargs)
end function $Worksheet_GetRange
```

```
subroutine Range_Select($object, $status)
 !dec$ attributes dllexport :: Range_Select
 !dec$ attributes value :: $object
 !dec$ attributes reference  :: $status
 implicit none
 integer(4), intent(in) :: $object                ! Указатель на объект
 integer(4), intent(out), optional :: $status     ! Статус метода
 integer(4) $$status, invokeargs
 invokeargs = AUTOAllocateInvokeArgs( )
 $$status = AUTOInvoke($object, 235, invokeargs)
 if(present($status)) $status = $$status
 call AUTODeallocateInvokeArgs(invokeargs)
end subroutine Range_Select

! $Workbook_GetCharts возвращает значение типа POINTER(p, INTEGER(4))
integer(4) function $Workbook_GetCharts($object, $status)
 !dec$ attributes dllexport :: $Workbook_GetCharts
 !dec$ attributes value :: $object
 !dec$ attributes reference :: $status
 implicit none
 integer(4), intent(in) :: $object                ! Указатель на объект
 integer(4), intent(out), optional :: $status     ! Статус метода
 integer(4) $$status, invokeargs
 integer(4), volatile :: $return
 invokeargs = AUTOAllocateInvokeArgs( )
 call AUTOAddArg(invokeargs, 'Charts', $return, .true., vt_dispatch)
 $$status = AUTOGetPropertyByID($object, 121, invokeargs)
 if(present($status)) $status = $$status
 $Workbook_GetCharts = $return
 call AUTODeallocateInvokeArgs(invokeargs)
end function $Workbook_GetCharts

! Charts_Add возвращает значение типа POINTER(p, INTEGER(4))
integer(4) function Charts_Add($object, Before, After, Count, $status)
 !dec$ attributes dllexport :: Charts_Add
 !dec$ attributes value :: $object
 !dec$ attributes reference :: Before
 !dec$ attributes reference :: After
 !dec$ attributes reference :: Count
 !dec$ attributes reference  :: $status
 implicit none
 integer(4), intent(in) :: $object                ! Указатель на объект
 type(variant), intent(in), optional :: Before, After, Count
 integer(4), intent(out), optional :: $status     ! Статус метода
 integer(4) $$status, invokeargs
 integer(4), volatile :: $return
 invokeargs = AUTOAllocateInvokeArgs( )
 ! Константы '$RETURN', '$ARGnn' записываются прописными буквами
 call AUTOAddArg(invokeargs, '$RETURN', $return, .true., vt_dispatch)
```

```
 if(present(Before)) call AUTOAddArg(invokeargs, '$ARG1', Before, .false.)
 if(present(After)) call AUTOAddArg(invokeargs, '$ARG2', After, .false.)
 if(present(Count)) call AUTOAddArg(invokeargs, '$ARG3', Count, .false.)
 $$status = AUTOInvoke($object, 181, invokeargs)
 if(present($status)) $status = $$status
 Charts_Add = $return
 call AUTODeallocateInvokeArgs(invokeargs)
end function Charts_Add
subroutine $Chart_ChartWizard($object, Source, Gallery, Format, PlotBy,          &
      CategoryLabels, SeriesLabels, HasLegend, Title, CategoryTitle, ValueTitle,  &
      ExtraTitle, $status)
 !dec$ attributes dllexport :: $Chart_ChartWizard
 !dec$ attributes value :: $object
 !dec$ attributes reference :: Source
 !dec$ attributes reference :: Gallery
 !dec$ attributes reference :: Format
 !dec$ attributes reference :: PlotBy
 !dec$ attributes reference :: CategoryLabels
 !dec$ attributes reference :: SeriesLabels
 !dec$ attributes reference :: HasLegend
 !dec$ attributes reference :: Title
 !dec$ attributes reference :: CategoryTitle
 !dec$ attributes reference :: ValueTitle
 !dec$ attributes reference :: ExtraTitle
 !dec$ attributes reference :: $status
 implicit none
 integer(4), intent(in) :: $object                  ! Указатель на объект
 type(variant), intent(in), optional :: Source, Gallery, Format, PlotBy, CategoryLabels, &
       SeriesLabels, HasLegend, Title,  CategoryTitle, ValueTitle, ExtraTitle
 integer(4), intent(out), optional :: $status       ! Статус метода
 integer(4) $$status, invokeargs
 invokeargs = AUTOAllocateInvokeArgs( )
 ! Константы '$ARGnn' записываются прописными буквами
 if(present(Source)) call AUTOAddArg(invokeargs, '$ARG1', Source, .false.)
 if(present(Gallery)) call AUTOAddArg(invokeargs, '$ARG2', Gallery, .false.)
 if(present(Format)) call AUTOAddArg(invokeargs, '$ARG3', Format, .false.)
 if(present(PlotBy)) call AUTOAddArg(invokeargs, '$ARG4', PlotBy, .false.)
 if(present(CategoryLabels)) call AUTOAddArg(invokeargs, '$ARG5',          &
      CategoryLabels, .false.)
 if(present(SeriesLabels)) call AUTOAddArg(invokeargs, '$ARG6',            &
      SeriesLabels, .false.)
 if(present(HasLegend)) call AUTOAddArg(invokeargs, '$ARG7', HasLegend, .false.)
 if(present(Title)) call AUTOAddArg(invokeargs, '$ARG8', Title, .false.)
 if(present(CategoryTitle)) call AUTOAddArg(invokeargs, '$ARG9',           &
     CategoryTitle, .false.)
 if(present(ValueTitle)) call AUTOAddArg(invokeargs, '$ARG10', ValueTitle, .false.)
 if(present(ExtraTitle)) call AUTOAddArg(invokeargs, '$ARG11', ExtraTitle, .false.)
```

```
  $$status = AUTOInvoke($object, 196, invokeargs)
  if(present($status)) $status = $$status
  call AUTODeallocateInvokeArgs(invokeargs)
 end subroutine $Chart_ChartWizard

 integer(4) function $Chart_Axes($object, Type, AxisGroup, $status)
  !dec$ attributes dllexport :: $Chart_Axes
  !dec$ attributes value :: $object
  !dec$ attributes reference :: Type
  !dec$ attributes reference :: AxisGroup
  !dec$ attributes reference :: $status
  implicit none
  integer(4), intent(in) :: $object                    ! Указатель на объект
  type(variant), intent(in) :: Type
  integer(4), intent(in) :: AxisGroup
  integer(4), intent(out), optional :: $status        ! Статус метода
  integer(4) $$status
  integer(4) invokeargs
  integer(4), volatile :: $return
  invokeargs = AUTOAllocateInvokeArgs( )
  ! Константы '$RETURN', '$ARG1', '$ARG2' записываются прописными буквами
  call AUTOAddArg(invokeargs, '$RETURN', $return, .true., vt_dispatch)
  call AUTOAddArg(invokeargs, '$ARG1', Type, .false.)
  call AUTOAddArg(invokeargs, '$ARG2', AxisGroup)
  $$status = AUTOInvoke($object, 23, invokeargs)
  if(present($status)) $status = $$status
  $Chart_Axes = $return
  call AUTODeallocateInvokeArgs(invokeargs)
 end function $Chart_Axes
end module excel97b
```

Замечание. Каждая процедура модуля EXCEL97B во второй строке содержит директиву

!dec$ attributes dllexport :: *имя процедуры*

которая обеспечивает создание LIB- и EXP-файлов, необходимых при генерации приложения, использующего DLL. Для работы рассматриваемого приложения эти директивы избыточны и могут быть удалены.

Приложение 1. Вывод русского текста в DOS-окно

Известно, что DOS- и Windows-коды букв русского алфавита различаются. Это обстоятельство надо учитывать при работе с консоль-приложениями FPS и CVF, поскольку присутствующие в программе символьные данные, если, конечно, исходные тексты набирались в программе, работающей под Windows, например в DS, имеют Windows-коды, а вывод текста, например, оператором

```
print *, 'Сообщение на русском языке'
```

выполняется в DOS-окно.

Так как в Фортране (FPS и CVF) при выводе в DOS-окно не выполняется автоматического преобразования русского Windows-текста в DOS-текст, то об этом должен позаботиться сам пользователь.

Также необходимость преобразований возникает при вводе оператором READ русского текста с клавиатуры или из DOS-файла: введенный текст имеет DOS-кодировку и для дальнейшей работы необходимо его преобразовать в Windows-текст.

Рассмотрим предпосылки создания программы, выполняющей преобразование DOS-текста в Windows-текст и обратное преобразование.

Пусть в файле work1.txt дана строка, содержащая символы как русского, так и латинского алфавитов.

Состав файла work1.txt.

```
Пример символьной строки. An example of a symbol string.
```

Заметим, что каждый символ данной, да и любой, строки имеет код - целое положительное число от 1 до 255. Существует также и *null*-символ, код которого равен нулю.

Задача 1. Найти сумму кодов всех символов строки файла work1.txt, за вычетом пробелов. Вывести также код каждого символа строки.

Используем при решении встроенную функцию IACHAR(*c*), которая возвращает значение стандартного целого типа, равное коду символа *c*. Тип параметра *c* - CHARACTER(1). Длину строки без концевых пробелов найдем функцией LEN_TRIM; *i*-й символ строки *string* - это *string*(*i*:*i*).

```
program symbol_codes
 integer(4) :: i, ico, sco                ! sco - искомая сумма кодов символов
 character(120) :: string
 character(1) :: ch
 open(10, file = 'work1.txt')             ! Подсоединяем файл к устройству В/В
 read(10, '(a)') string                   ! Ввод строки (одной записи) файла
 do i = 1, len_trim(string)               ! LEN_TRIM(string) - возвращает длину
```

```
  ch = string(i:i); ico = iachar(ch)        ! строки без концевых пробелов
  if(ch /= ' ') sco = sco + ico             ! Сумма кодов, не включая коды пробелов
  print *, 'Code of symbol ', ch, ' = ', ico
  read *                                    ! Ожидаем нажатия Enter
 end do
 print *, 'Сумма кодов символов строки без кодов пробелов sco = ', sco
end program symbol_codes
```

Просматривая выводимые данные, мы обнаружим, что пробел имеет код 32, а точка - 46. Буквы английского алфавита имеют код в диапазоне от IACHAR('A') = 65 до IACHAR('z') = 122. Буквы русского алфавита в случае DOS-кодовой страницы 866 имеют код в диапазоне от IACHAR('A') =

= 128 до IACHAR('я') = 239. То есть прописные буквы алфавита имеют меньший код, чем соответствующие им строчные буквы.

Из сопоставления минимального (128) и максимального кодов (239) букв следует, что в этом диапазоне кодов находятся не только коды русских букв, но и коды иных символов. Коды русских букв в DOS-кодовой странице 866 изменяются в диапазонах:

- 128-159 - коды прописных букв от *А* до *Я*;
- 160-175 - коды строчных букв от *а* до *п*;
- 224-239 - коды строчных букв от *р* до *я*.

Можно, применив встроенную функцию CHAR(*i*), выполнить и обратное преобразование: код символа - символ.

Задача 2. Вывести все буквы русского алфавита.

```
program russian_letters
 integer(4) :: i
 character(1) :: ch
 do i = 128, 128 + 31                ! Вывод прописных букв
  print '(a, i3, a, a)', 'Capital latter with code ', i, ' - ', char(i)
  read *                             ! Ожидаем нажатия Enter
 end do
 do i = 160, 160 + 15                ! Вывод первых 16 строчных букв
  print '(a, i3, a, a)', 'Small letter with code ', i, ' - ', char(i)
  read *                             ! Ожидаем нажатия Enter
 end do
 do i = 224, 239                     ! Вывод следующих 16 строчных букв
  print '(a, i3, a, a)', 'Small letter with code ', i, ' - ', char(i)
  read *                             ! Ожидаем нажатия Enter
 end do
end program russian_letters
```

Сохраним после ввода с клавиатуры строку текста в файле, использовав, например, такую программу:

```
program string_to_file
 character(120) string
 open(10, file = 'work1.txt')        ! Подсоединяем файл к устройству В/В
 read '(a)', string                  ! Введем: Пример DOS-строки текста.
 write(10, '(a)') string             ! Вывод DOS-строки в файл work1.txt
end program string_to_file
```

Откроем файл work1.txt, например, в DS или в стандартной Windows-программе NotePad (блокнот). Тогда введенная в DOS-режиме строка файла work1.txt предстанет в виде нечитаемого набора символов:

ЦаЁ¬Ґa DOS-бва®ЄЁ вЃЄбв .

Задача 3. Преобразовать строку из DOS-представления в Windows-представление.

Задача 4. Преобразовать строку из Windows-представления в DOS-представление.

Решим прежде промежуточную задачу.

Задача 5. Вывести в файл work2.txt Windows-коды букв русского алфавита, принимая во внимание, что Windows-код русской буквы больше ее DOS-кода.

Воспользуемся для этого программой

```
program windows_codes
 integer(2) :: i
 open(11, file = 'work2.txt')          ! Подсоединяем файл к устройству В/В
 do i = 160, 255
   write(11, '(i4, 2x, a1)') i, char(i)  ! Вывод Windows-кодов и символов
 end do                                ! в файл work2.txt
end program windows_codes
```

Проанализировав файл work2.txt, мы обнаружим, что коды русских букв в Windows-кодовой странице 1251 изменяются в таких диапазонах:

- 192-223 - коды прописных букв от *А* до *Я*;
- 224-255 - коды строчных букв от *а* до *я*.

Таким образом, чтобы преобразовать DOS-букву *ru_letter* русского алфавита в Windows-букву *ru_letter* русского алфавита потребуется выполнить для букв с DOS-кодами от 128 до 175 (буквы *А* - *Я*, *а* - *п*) оператор

ru_letter = CHAR(*ru_letter* + (192 - 128))

А для букв с DOS-кодами от 224 до 239 (буквы *р* - *я*) следует применить оператор

ru_letter = CHAR(*ru_letter* + (255 - 239))

Понятно, что обратное преобразование потребует уменьшение Windows-кода буквы до соответствующего ее DOS-кода.

Оформим преобразования DOS - Windows и Windows - DOS в виде внешней символьной функции

string = RuDosWin(*string*, *dos_win*)

которую разместим в модуле TextTransfer. Функция RuDosWin такова, что если ее параметр *dos_win* равен .TRUE., то выполняется преобразование DOS - Windows, в противном случае (*dos_win* = .FALSE.) выполняется преобразование Windows - DOS.

Функция RuDosWin позволит выводить в консоль-проектах русские тексты в DOS-окно и читать оператором READ русские DOS-тексты в приложениях CVF и FPS.

```
module TextTransfer
! Длина строк - результирующих переменных символьных функций
 integer(4), parameter :: ncresults = 250

 contains

! Функция преобразовывает текст DOS в текст Windows, если dos_win = .TRUE.,
! и преобразовывает текст Windows в текст DOS, если dos_win = .FASLE.
 function RuDosWin(string, dos_win)  ! Длина строки string не должна превышать
  character(ncresults) :: RuDosWin      ! ncresults символов
  ! Параметры string и dos_win имеют вид связи IN
  ! и, следовательно, не должны изменяться в RuDosWin
  character(*), intent(in) :: string
  logical(4), intent(in) :: dos_win
  ! dif - величина, на которую при преобразовании изменяется код буквы
  integer(2) :: i, dos_win_code, dif
   RuDosWin = string
  do i = 1, len_trim(RuDosWin)
   ! dos_win_code – DOS- или Windows-код символа
   dos_win_code = iachar(RuDosWin(i:i))
   dif = 0                          ! dif больше нуля, если символ - русская буква
   if(dos_win) then                 ! Если преобразование DOS - Windows
    select case(dos_win_code)       ! Найдем величину dif
    case(128 : 175)                 ! DOS-русские буквы от А до Я и от а до п
      dif = 64
    case(224 : 239)                 ! DOS-русские буквы от р до я
      dif = 16
    end select
   else                             ! Преобразование Windows - DOS
    select case(dos_win_code)
    case(192 : 239)                 ! Windows-русские буквы от А до Я и от а до п
      dif = -64
    case(240 : 255)                 ! Windows-русские буквы от р до я
      dif = -16
    end select
   end if
  ! Выполняем преобразование символа, если он является буквой русского алфавита
```

```
   if(dif /= 0) RuDosWin(i:i) = char(dos_win_code + dif)
  end do
 end function RuDosWin
end module TextTransfer
```

Пример. Вывести в консоль-приложении заданный в программе русский текст (он имеет Windows-кодировку) в DOS-окно, а введенный с клавиатуры DOS-текст вывести в текстовой файл a.txt, выполнив предварительно преобразование DOS-Windows.

```
program text_go
 use TextTransfer
 character(120) :: string
 ! Выводим на консоль DOS-текст Введите строку на русском языке,
 ! получаемый после преобразования Windows-DOS
 ! Встроенная функция TRIM выполняет отсечение концевых пробелов
 print *, trim(RuDosWin('Введите строку на русском языке', .false.))
 read(*, '(a)') string                ! Вводим с клавиатуры DOS-текст
 ! Введем: Текст на русском языке
 print *, string                      ! Выводим строку на консоль без преобразований
 ! Результат: Текст на русском языке
 open(10, file = 'a.txt')
 write(10, '(a)') string              ! Выводим строку в файл a.txt без преобразований
 ! Просмотрим файл a.txt в "Блокноте" (NotePad)
 ! Результат - нечитаемый текст: ’Ґбв -  агббЄ®¬ п§лЄЃ
 ! Выводим строку в файл a.txt после преобразований
 write(10, '(a)') RuDosWin(string, .true.)
 ! Просмотрим файл a.txt в "Блокноте"
 ! Результат: Текст на русском языке
end program text_go
```

Приложение 2. Нерекомендуемые, устаревшие и исключенные свойства Фортрана

Стандарт Фортран 90 сохранил все свойства Фортрана 66 и Фортрана 77. Теперь с введением новых средств для достижения одного и того же результата язык нередко предоставляет пользователю несколько возможностей. Причем часть из них стандарт относит к избыточным или устаревшим. Помимо этого, прежние стандарты содержат свойства, применение которых ухудшает структуру программы: операторы EQUIVALENCE, ENTRY (разд. 8.20) и вычисляемый GOTO. Эти операторы включены в стандарт Фортран 90, но относятся к нерекомендуемым.

Устаревшие свойства Фортрана не могут быть рекомендованы к применению также и потому, что следующий стандарт может их просто не содержать.

П.-2.1. Нерекомендуемые свойства Фортрана

П.-2.1.1. Фиксированная форма записи исходного кода

По умолчанию длина строки исходного текста при записи его в фиксированной форме равна 72 символам. Однако в результате применения директивы $FIXEDFORMLINESIZE она может быть увеличена до 80 или 132 символов.

Интерпретация символов строки Фортран-программы в фиксированной форме зависит от того, в какой колонке они указаны. Правила интерпретации сведены в табл. П.-2.1.

Таблица П.-2.1. Интерпретация символов строки программы

Колонки	*Интерпретация символов*
1	Символы $ или !MS$ (указывает на директиву)
1	Символы *, или c, или C, или ! (указывает на комментарий)
1-5	Метка оператора
6	Символ продолжения (кроме нуля и пробела)
7-72	Оператор Фортрана
73 и выше	Игнорируются

В фиксированной форме различают 5 типов строк: *комментарии*, *начальные строки*, *строки продолжения*, *директивы* и *отладочные строки*.

Комментарий располагается либо после символа * и латинских букв *c* или *C*, размещенных в первую позицию строки, либо после восклицательного

знака, размещаемого в любой (с 1-й по 72-ю) позиции строки. Строка с восклицательным знаком в колонке 6 интерпретируется как строка продолжения. Комментарии не оказывают никакого влияния на работу программы.

Первая, или единственная, строка оператора Фортрана называется *начальной строкой*. Начальная строка имеет либо пробел, либо 0 в 6-й колонке; в колонках 1-5 указываются метка оператора либо пробелы.

Метка оператора - целая константа без знака в диапазоне от 1 до 99999. Метки используются для ссылки на оператор. Ссылка на оператор выполняется в операторах перехода (GOTO), в операторах В/В для ссылки на оператор FORMAT, в операторах цикла и в других случаях.

Пример:

```
      real x / -1.32 /, y / 6.487 /
      write(*, 1) x, y                    ! Ссылка на оператор format
1     format(2x, 'x = ', f6.2, 2x, 'y = ', f6.3)
      ...
      if(exp(x/3.) .gt. cmax) go to 89    ! Ссылка на оператор return
      ...
89    return
      ...
```

Строка продолжения содержит пробелы в колонках 1-5 и символ (отличный от нуля или пробела) в колонке 6. Строка продолжения увеличивает число доступных для записи оператора позиций. Число строк продолжения ограничено доступной памятью ЭВМ.

Директивы управляют работой компилятора. Директива начинается с символа $, или префикса !MS$ в FPS, или префикса !DEC$ в CVF. В любом случае в фиксированной форме директива должна начинаться в первой колонке строки, например:

```
$NOFREEFORM
!DEC$NOFREEFORM
!MS$NOFREEFORM
```

Отладочные строки. Включение режима проверки осуществляется директивой $DEBUG. Выключение - директивой $NODEBUG.

В одной строке исходного текста могут располагаться строки двух типов:

- начальная строка и комментарий (после символа !);
- строка продолжения и комментарий (после символа !).

Пример двух вариантов одной и той же программы, содержащей начальные строки, комментарии и в варианте 2 строку продолжения.

```
C Вариант 1
C234567 - нумерация позиций
      PROGRAM p1          ! Комментарий
      implicit none       ! Должны быть объявлены типы всех объектов данных
```

```
      real d /4./, s /2.3/        ! Объявляем переменные
      write(*, *) s / sqrt(d)     ! SQRT(x) - встроенная функция
      end                         ! вычисления квадратного корня из x
* Вариант 2
*234567 - Комментарий
c Это тоже комментарий
      data d /4./, s /2.3/        ! Инициализация переменных
      write(*, *)                 ! Начальная строка
     *    s/sqrt(d)               ! Строка продолжения
      end
```

Пояснение. Отказаться от раздела объявлений во втором варианте позволяют существующие умолчания о типах данных.

П.-2.1.2. Оператор EQUIVALENCE

Оператор указывает, что две или более переменные занимают одну и ту же область памяти.

EQUIVALENCE(*nlist*) [,(*nlist*)] ...

nlist - список двух или более переменных (простых или составных), разделенных запятыми. Список не может включать формальные параметры, динамические массивы и ссылки. Размерности массивов списка должны быть целыми константами. Заданное без размерностей имя массива адресует первый элемент массива.

Переменные, адресующие одну и ту же область памяти, называются ассоциированными по памяти. При этом не выполняется никакого автоматического преобразования типов данных. Ассоциированные символьные элементы могут перекрываться, что иллюстрируется следующим примером:

```
character a*4, b*4, c*3(2)
equivalence(a, c(1)), (b, c(2))
```

Графически перекрытие отображено на рис. П.-2.1.

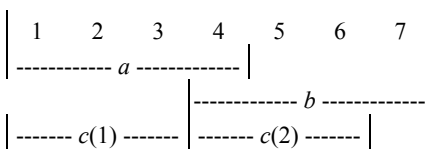


Рис. П.-2.1. Перекрытие ассоциированных символьных элементов

Правила ассоциирования элементов:

- переменная не может занимать более одной области памяти. Так, следующие операторы вызовут ошибку, поскольку пытаются адресовать переменную *r* двум различным областям памяти, в одной из которых размещен элемент *s*(1), а в другой - *s*(2):

```
real r, s(10)
equivalence(r, s(1))
equivalence(r, s(2))          ! Ошибка
```

- соответствующие элементы массивов должны ассоциироваться последовательно. Так, следующие операторы вызовут ошибку:

```
real r(10), s(10)
equivalence(r(1), s(1))
equivalence(r(5), s(7))       ! Допустимо: equivalence(r(5), s(5))
```

Компилятор всегда выравнивает несимвольные элементы по четному байту. Символьные и несимвольные элементы могут быть ассоциированы, если несимвольный элемент начинается на четном байте. При необходимости компилятор разместит символьный элемент так, чтобы несимвольный начинался на четном байте. Однако это не всегда возможно. В следующем примере невозможно разместить символьный массив так, чтобы оба несимвольных элемента начинались на четном байте памяти.

```
character*1 char1(10)
real a, b
equivalence(a, char1(1))
equivalence(b, char1(2))
```

Элемент списка *nlist* нельзя инициализировать в операторах объявления типа. Пример ошибки:

```
integer i /1/, j              ! Ошибка
equivalence(i, j)
```

Оператор EQUIVALENCE не может разделять память между двумя *common*-областями или между элементами одного и того же общего блока.

Оператор EQUIVALENCE может расширить *common*-область в результате добавления переменных, продолжающих *common*-область. Нельзя расширять *common*-область, добавляя элементы перед этой областью. Так, следующие операторы вызовут ошибку:

```
common /b1/ r(10)
real s(20)
equivalence(r(1), s(7))       ! Правильно: equivalence(r(1), s(1))
```

Ошибка возникает из-за того, что перед общей областью должны быть размещены элементы *s*(1), ..., *s*(6).

Если использована директива $STRICT или опция компилятора /4Ys, то при работе с оператором EQUIVALENCE следует придерживаться следующих правил:

- если ассоциируемый объект имеет стандартный целый, логический, вещественный, вещественный двойной точности тип или относится к упорядоченному производному типу с числовыми и логическими компонентами, то все объекты в операторе EQUIVALENCE должны иметь один из таких типов;

- если ассоциируемый объект имеет символьный тип или относится к упорядоченному производному типу с символьными компонентами, то все объекты в операторе EQUIVALENCE должны иметь один из таких типов, хотя могут быть и разной длины;
- если ассоциируемый объект имеет упорядоченный производный тип, который не является чисто числовым или символьным, то все объекты в операторе EQUIVALENCE должны относиться к тому же производному типу;
- если ассоциируемый объект имеет встроенный, но нестандартный тип, например INTEGER(1), то все объекты в операторе EQUIVALENCE должны иметь тот же тип и параметр разновидности типа.

Оператор EQUIVALENCE предназначен для экономии оперативной памяти. Так, можно использовать одну и ту же память под символьный и вещественный массивы, уменьшая издержки памяти в два раза. Недостатки оператора очевидны: если переменные *a* и *b* занимают одну и ту же память, то изменение значения одной из переменных приведет к изменению значения и другой. Поэтому каждый раз, применяя переменную *a*, необходимо следить, чтобы ее значение не было случайным образом изменено в результате изменения значения переменной *b*. В больших программах это может оказаться весьма затруднительным.

Другие применения оператора - создание псевдонимов и отображение одного типа данных в другой, что можно было бы использовать при хранении и выборке данных. Теперь все эти задачи можно решить более надежными и удобными средствами. В зависимости от ситуации вместо оператора EQUIVALENCE можно использовать автоматические массивы, размещаемые массивы и ссылки для повторного использования памяти; ссылки как псевдонимы и функцию TRANSFER (разд. 6.5) для отображения одного типа данных в другой.

П.-2.1.3. Оператор ENTRY

Рассмотренный в разд. 8.12 оператор ENTRY дополнительного входа в процедуру не может быть рекомендован для применения, как оператор, серьезно ухудшающий структуру программы.

П.-2.1.4. Вычисляемый GOTO

Оператор имеет вид:

GOTO (*labels*) [,] *n*

labels - список одной или более разделенных запятыми меток исполняемых операторов того же блока видимости. Одна и та же метка может появляться в списке более одного раза.

n - целочисленное выражение.

Оператор передает управление *n*-й метке списка *labels*. Допустимый диапазон значений *n*: $1 \le n \le m$, где *m* - число меток в списке *labels*. Если *n* выходит за границы допустимого диапазона, то вычисляемый GOTO работает так же, как и пустой оператор CONTINUE. Переход внутрь DO-, IF-, SELECT CASE- или WHERE-конструкций запрещен.

Пример:

```
next = 1
goto (10, 20) next                ! Передача управления на оператор 10 continue
...
10 continue
...
20 continue
```

Оператор заменяется конструкциями IF и SELECT CASE.

П.-2.1.5. Положение оператора DATA

Оператор DATA задания начальных значений переменных (разд. 3.7) можно располагать среди исполняемых операторов программы. Но этой возможностью пользоваться не рекомендуется. Следует располагать операторы DATA в разделе описаний перед первым исполняемым оператором.

П.-2.2. Устаревшие свойства Фортрана, определенные стандартом 1990 г.

П.-2.2.1. Арифметический IF

Оператор имеет вид:

IF(*expr*) *m*1, *m*2, *m*3

expr - целочисленное или вещественное выражение.

*m*1, *m*2, *m*3 - метки исполняемых операторов того же блока видимости. Значения меток могут совпадать.

Оператор обеспечивает переход по метке *m*1, если *expr* < 0, по метке *m*2, если *expr* = 0, и по метке *m*3, если *expr* > 0.

Пример. Вычислить число отрицательных, нулевых и положительных элементов целочисленного массива.

```
integer a(10) /-2, 0, 3, -2, 3, 3, 3, 0, 4, 0/
integer k1, k2, k3
k1 = 0; k2 = 0; k3 = 0
do i = 1, 10
 if(a(i)) 10, 11, 12
 10 k1 = k1 + 1; cycle
 11 k2 = k2 + 1; cycle
 12 k3 = k3 + 1
end do
```

```
print *, k1, k2, k3                ! 2 3 5
end
```

Оператор заменяется конструкциями IF и SELECT CASE.

П.-2.2.2. Оператор ASSIGN присваивания меток

Оператор присваивания меток имеет вид:

ASSIGN *метка* TO *имя переменной*

метка - целое число из диапазона 1-99999.

В результате выполнения оператора ASSIGN переменной, которая должна быть целого типа, будет присвоено значение *метки*.

Пример:

```
integer label, k
y = (2.5 * sin(5.2))**3
...                                ! k получает некоторое значение
if(k .eq. 1) assign 89 to label
write(*, label) y                  ! Ссылка на оператор format
89 format(2x, 'y = ', f10.3)
assign 17 to label
if(y .gt. 0) go to label           ! Переход к метке 17
...
17 stop
...
```

Переменная, получившая в результате выполнения оператора ASSIGN значение метки, не может использоваться как переменная, имеющая численное значение, в выражениях Фортрана. Переменная, получившая значение помимо оператора ASSIGN, например в результате присваивания, не может использоваться в операторе перехода. Также в операторе GOTO не может быть использована и именованная константа.

Оператор используется для выбора подходящего оператора FORMAT и в назначаемом операторе GOTO. В первом случае заменой оператору ASSIGN является задание формата при помощи символьных выражений (разд. 9.2). Назначаемый GOTO всегда может быть успешно заменен конструкциями IF или SELECT CASE.

П.-2.2.3. Назначаемый GOTO

Оператор имеет вид:

GOTO *var* [[,] (*labels*)]

var - переменная целого типа, значением которой является метка исполняемого оператора. Значение переменной *var* должно быть определено оператором ASSIGN в том же блоке видимости.

labels - список одной или более разделенных запятыми меток исполняемых операторов того же блока видимости. Одна и та же метка может появляться в списке более одного раза.

Оператор передает управление оператору, метка которого совпадает со значением *var*. Запрещается переход внутрь DO-, IF-, SELECT CASE- и WHERE-конструкций.

Пример:

```
integer vi, cle
...                              ! cle получает некоторое значение
if(cle .eq. 1) then              ! Назначим vi нужное значение метки
  assign 200 to vi
else if(cle .eq. 2) then
  assign 400 to vi
else
  assign 100 to vi
end if
goto vi (100, 200, 400)
100 continue
...
goto 500
200 continue
...
goto 500
400 continue
...
500 continue
end
```

П.-2.2.4. Варианты DO-цикла

Параметр DO-цикла и циклического списка, а также выражения, задающие пределы и шаг изменения параметра, могут быть вещественного типа (REAL(4) или REAL(8)). Например:

```
do x = 0.4, 20.4, 0.4
  ...
end do
```

Для приведенного цикла естественно ожидать, что число итераций *ni*, которое вычисляется по формуле

$$ni = \mathrm{MAX}(\mathrm{INT}((stop - start + inc)/inc), 0),$$

будет равно 51. На самом деле в результате ошибок округления промежуточным результатом при расчете *ni* может явиться число 50.999999..., а не 51.000000... Тогда после применения функции *int* значение *ni* будет равно 50. Поскольку такое может случиться, то применение вещественного параметра в DO-цикле и циклическом списке нежелательно.

DO-цикл с вещественным параметром заменяется DO- или DO WHILE-циклом.

Не может быть рекомендовано и завершение вложенного DO-цикла одним общим помеченным оператором, например:

```
s = 0.0
do 1 i = 1, n1
   do 1 j = 1, n2
      do 1 k = 1, n3
         s = s + a(i, j, k)
1     continue
```

Такая форма может послужить источником разнообразных ошибок.

П.-2.2.5. Переход на END IF

Переход на END IF может быть выполнен не только из конструкции, которую он завершает, но и извне. Этого следует избегать и пользоваться переходом на следующий за END IF оператор.

П.-2.2.6. Альтернативный возврат

Использование альтернативного возврата из подпрограммы (разд. 8.19) ухудшает структуру программы. Вместо него можно при выходе из подпрограммы вернуть код ее завершения, а последующее ветвление выполнить, например, конструкцией SELECT CASE.

П.-2.2.7. Дескриптор формата H

Дескриптор рассмотрен в разд. 9.7. Вместо него лучше использовать преобразование апострофа или кавычек. Так, вместо

```
write(*, '(2x, 31HВведите границы отрезка [a, b]:)')
```

лучше записать:

```
write(*, '(2x, a)') 'Введите границы отрезка [a, b]: '
```

или

```
write(*, "(2x, 'Введите границы отрезка [a, b]: ')")
```

В отличие от первого оператора два последних не требуют подсчета числа передаваемых символов.

П.-2.3. Устаревшие свойства Фортрана, определенные стандартом 1995 г.

Приводимые ниже устаревшие свойства языка будут удалены из Фортрана в последующих версиях и, следовательно, их применение нежелательно. В приводимом перечне тремя звездочками (***) отмечены свойства, которые классифицируются как устаревшие, начиная с версии

Фортран 95. Прочие свойства отнесены к устаревшим еще стандартом Фортран 90. В списке 9 устаревших свойств:

- арифметический оператор IF;
- завершение нескольких DO-циклов одним оператором и завершение DO-цикла оператором, отличным от CONTINUE или END DO;
- альтернативный возврат из процедуры;
- вычисляемый оператор GOTO (***);
- операторная функция (***);
- размещение оператора DATA среди исполняемых операторов (***);
- символьные функции предполагаемой - CHARACTER(*len* = *) - длины (***);
- фиксированная форма исходного кода (***);
- форма CHARACTER* для объявления символьных типов данных (***).

Предполагается изъять из Фортрана первые 6 свойств уже в следующем стандарте.

П.-2.4. Исключенные свойства Фортрана

Стандарт 1995 г. исключил из Фортрана:

1) DO-цикл с вещественным и двойной точности параметром, например запрещен цикл:

```
real(4) :: x, xs = 1.0, xf = 3.0, dx = 0.1
do x = xs, xf, df
  print *, x * sin(x)
end do
```

2) переход на END IF из внешнего блока;
3) оператор PAUSE;
4) оператор ASSIGN присваивания меток и назначаемый GO TO; ясно, что нельзя использовать в качестве метки целочисленную переменную, получившую значение в результате выполнения оператора ASSIGN;
5) символьные константы с *указателем длины*, называемые также холлеритовскими константами.

Приложение 3. Дополнительные процедуры

CVF и FPS, помимо предусмотренных стандартом Фортрана встроенных процедур, содержат большое число дополнительных процедур, которые в документации распределены по разделам:

Раздел	*Модуль*
Запуск программ	DFLIB (MSFLIB)
Управление программой	"
Работа с системой, дисками и директориями	"
Управление файлами	"
Случайные числа	"
Процедуры даты и времени	"
Процедуры клавиатуры и звука	"
Обработка ошибок	"
Аргументы в командной строке	"
Сортировка и поиск в массиве	"
Процедуры управления операциями с плавающей точкой	"
Процедуры QuickWin	"
Графические процедуры	"
Создание диалогов	DIALOGM
Работа с национальным языком	DFNLS (MSFNLS)
Обеспечение совместимости с другими платформами	PORTLIB

Перечислим и опишем назначение дополнительных процедур (кроме процедур QuickWin и графических процедур), интерфейс к которым находится в модуле MSFLIB. Для вызова таких процедур необходимо к вызывающей программной единице подключить этот модуль, применив оператор USE MSFLIB. Графические процедуры подробно рассмотрены в гл. 12. Там же рассмотрена часть процедур QuickWin.

П.-3.1. Запуск программ

Выполняет функция RUNQQ; вызывает другую программу и ожидает ее завершения.

П.-3.2. Управление программой

Функции RAISEQQ и SIGNALQQ используются для обработки прерываний операционной системы с целью управления исполнением пользовательской программы. Процедура SLEEPQQ - подпрограмма, остальные - функции.

Процедура	*Назначение*
RAISEQQ	Передает сигнал прерывания выполняемой программе, моделирующий прерывание операционной системы
SIGNALQQ	Управляет обработкой сигналов
CALL SLEEPQQ	Задерживает исполнение программы на указанный в миллисекундах промежуток времени

Пример:

```
program sleep
 use dflib                    ! Обязательная ссылка на модуль DFLIB
 print *, 'Before sleep'
 call sleepqq(500)            ! Задержка на 500 миллисекунд
 print *, 'After sleep'
end program sleep
```

П.-3.3. Работа с системой, дисками и директориями

Процедуры используются для работы с физическими устройствами, директориями и для идентификации полных имен путей. Процедура SETENVQQ - подпрограмма, остальные - функции.

Процедура	*Назначение*
CHANGEDIRQQ	Делает указанную директорию текущей или установленной по умолчанию
CHANGEDRIVEQQ	Делает указанный диск текущим
DELDIRQQ	Удаляет указанную директорию
GETDRIVESIZEQQ	Возвращает размер указанного диска
GETDRIVESQQ	Возвращает имена имеющихся в системе дисков
MAKEDIRQQ	Создает новую директорию с указанным именем
GETENVQQ	Получает значение из текущего окружения
CALL SETENVQQ	Добавляет новую переменную окружения или устанавливает значение существующей переменной
SYSTEMQQ	Выполняет команду путем передачи командной строки интерпретатору команд операционной системы

Пример:

```
program drive_dir
 use dflib
 character(40) :: temp
 integer(4) :: fres, total, available
 logical(4) :: sres
 temp = getdrivesqq( )
 print *, trim(temp)                ! A CDE
 ! Функция GETDRIVESIZEQQ в случае успеха вернет .TRUE.
 sres = getdrivesizeqq(temp(3:3), total, available)
 ! Общее и свободное число байт на диске C:
 print '(2i15)', total, available      !      1099956224      256737280
 ! Создаем переменную окружения path2
 sres = setenvqq('path2=c:\')       ! строку 'path2=c:\' записываем без пробелов
 fres = getenvqq('path2', temp)
 print *, temp                      ! c:\mine
 temp = 'c:\'                       ! Будем искать рабочую директорию на диске c:\
 fres = getdrivedirqq(temp)         ! Поиск рабочей директории
 print *, temp                      ! C:\Program Files\ArrayVisualizer\SAMPLES
 sres = makedirqq('c:\temp2')       ! Создаем директорию c:\temp2
 sres = changedirqq('c:\temp2')     ! Делаем директорию c:\temp2 рабочей
 temp = 'c:\'                       ! Ищем рабочую директорию на диске c:\
 fres = getdrivedirqq(temp)
 print *, temp                      ! c:\temp2
 sres = deldirqq('c:\temp2')        ! Удаляем директорию c:\temp2
end program drive_dir
```

П.-3.4. Управление файлами

Процедуры используются для управления файлами и получения информации, хранящейся о файлах в операционной системе. Процедуры PACKTIMEQQ и UNPACKTIMEQQ - подпрограммы, остальные - функции.

Процедура	*Назначение*
DELFILESQQ	Удаляет указанные файлы в указанной директории
FINDFILEQQ	Ищет указанный файл в директориях, содержащихся в переменной окружения
FULLPATHQQ	Возвращает полное имя для указанного файла или директории
GETDRIVEDIRQQ	Возвращает текущий диск и путь к текущей директории
GETFILEINFOQQ	Возвращает информацию о файлах, имена которых содержат заданную строку
CALL PACKTIMEQQ	Упаковывает время для использования функцией SETFILETIMEQQ
RENAMEFILEQQ	Изменяет старое имя файла на новое

SETFILEACCESSQQ	Устанавливает способ доступа к заданному файлу
SETFILETIMEQQ	Устанавливает дату изменения для указанного файла
SPLITPATHQQ	Выделяет в полном имени файла его 4 компонента
CALL UNPACKTIMEQQ	Распаковывает упакованное время и дату в отдельные компоненты

Пример. Открываем файл c:\test.txt как новый, но прежде выполняется попытка его удаления функцией DELFILESQQ.

```
program files
 use dflib
 character(40) :: fn = 'c:\test.txt', pathname
 character(40) :: directory, file
 character(4) :: drive, extension
 integer(4) :: ios, fres
 logical(4) :: sres
 fres = delfilesqq(fn)              ! Удаляем файл fn, если он существует
 sres = setenvqq('path=c:\')        ! Устанавливаем переменную окружения path
 ! Функция FINDFILEQQ вернет нуль, если файл не найден
 fres = findfileqq('test.txt', 'path', pathname)
 ! Подсоединяем файл fn к устройству 10
 open(10, file = fn, status = 'new', iostat = ios)
 if(ios == 0) then                  ! ios равен нулю, если файл удалось открыть
   ! Функция FINDFILEQQ возвращает длину полного пути к файлу
   fres = findfileqq('test.txt', 'path', pathname)
   print *, 'Full path to file test.txt: ', pathname! C:\test.txt
 else
   stop 'Cannot open file c:\test.txt'
 end if
 sres = renamefileqq('c:\test.txt', 'c:\test2.txt')! Изменяем имя файла
 sres = splitpathqq(pathname, drive, directory, file, extension)
 print *, 'path = ', trim(pathname), '; drive = ', trim(drive)! path = c:\test.txt; drive = c:
 print *, 'directory = ', trim(directory), '; file = ', trim(file)! directoty = \; file = test2
 print *, 'extension = ', trim(extension)       ! extension = .txt
 fres = delfilesqq('c:\test2.txt')              ! Удаляем файл 'c:\test2.txt'
end program files
```

П.-3.5. Генерация случайных чисел

Помимо предусмотренных стандартом подпрограмм (разд. 6.19)

CALL RANDOM_NUMBER и CALL RANDOM_SEED

для получения случайных чисел можно использовать следующие подпрограммы:

Подпрограмма	*Назначение*

CALL RANDOM	Возвращает псевдослучайное вещественное число, большее или равное нулю и меньшее единицы
CALL SEED	Изменяет начальную точку генератора псевдослучайных чисел

Пример:

```
program rnd_seed
 use dflib
 real(4) :: rnd
 integer(4) :: seedvalue = 1025
 call seed(seedvalue)              ! Затравка датчика случайных чисел
 call random(rnd)                  ! Первое случайное число
 print *, rnd                      !    2.303988E-02
 call seed(2*seedvalue)            ! Новая затравка датчика случайных чисел
 call random(rnd)                  ! Первое случайное число
 print *, rnd                      !    4.607977E-02
end program rnd_seed
```

П.-3.6. Управление датой и временем

Процедуры используются для управления системными датой и временем.

Процедура	*Тип*	*Назначение*
CALL GETDAT	Подпрограмма	Возвращает системную дату
CALL GETTIM	"	" системное время
SETDAT	Функция	Установка даты
SETTIM	"	" времени

Пример:

```
program date_time
 use dflib
 ! Год, месяц, число
 integer(2) :: year, month, date
 ! Час, минуты, секунды, сотые доли секунд
 integer(2) :: hours, minutes, seconds, seconds100
 logical(4) :: sres
 call getdat(year, month, date)
 call gettim(hours, minutes, seconds, seconds100)
 print *, year, month, date           ! 2000   12   15
 print *, hours, minutes, seconds, seconds100!     21   12   59   28
 sres = setdat(year, month, date + 4_2)          ! Изменяем число
 sres = settim(hours - 1_2, minutes, seconds, seconds100)! Изменяем час
end program date_time
```

П.-3.7. Ввод с клавиатуры и генерация звука

Процедуры используются для непосредственного чтения с клавиатуры, минуя систему ввода/вывода Фортрана, и воспроизведения звуковых сигналов. Процедура BEEPQQ - подпрограмма, остальные - функции.

Процедура	*Назначение*
CALL BEEPQQ	Воспроизводит сигнал с заданными в миллисекундах продолжительностью и частотой
GETCHARQQ	Возвращает введенный символ
GETSTRQQ	Читает символьную строку с клавиатуры, используя буфер
PEEKCHARQQ	Проверяет, была ли нажата какая-либо клавиша консоли

Пример:

```
program beep_char
 use dflib
! Частота (в Гц) и продолжительность в миллисекундах сигнала
 integer(4) :: frequency = 500, duration = 1000
 character(1) :: char
 character(120) :: string
 integer(4) :: length
 logical(4) :: pressed
 call beepqq(frequency, duration)              ! Звуковой сигнал с частотой 500 Гц
 print *, 'Shall I stop a program (press Y or N)? '
 char = getcharqq( )
 read *
 if(char == 'Y' .or. char == 'y') stop 'Program is terminated by a user'
 print *, 'Enter a test string '
 length = getstrqq(string)          ! Возвращает длину строки string
 if(length > 0) then
   print *, trim(string)
 else
   print *, 'Missing value'
 end if
 pressed = .false.                  ! Демонстрация функции PEEKCHARQQ
 do                                 ! Вывод сообщения продолжается до тех пор,
   print *, 'Press any key to EXIT DO-loop'   ! пока не нажата какая-либо клавиша
   pressed = peekcharqq( )          ! Функция вернет .TRUE., если нажата клавиша
   if(pressed) exit                 ! Клавиша нажата - выход из цикла
 end do
 print *, 'Key is pressed'
 read *
end program beep_char
```

П.-3.8. Обработка ошибок

Процедуры управляют обработкой критических, приводящих к завершению программы ошибок, и позволяют получить дополнительную информацию о причинах иных ошибок. Процедура GETLASTERRORQQ - функция, остальные - подпрограммы.

Процедура	*Назначение*
GETLASTERRORQQ	Возвращает последнюю обнаруженную дополнительной процедурой ошибку
CALL MATHERRQQ	Заменяет обработку ошибок по умолчанию на обработку встроенными функциями
CALL SETERRORMODEQQ	Устанавливает способ обработки критических ошибок

П.-3.9. Аргументы в командной строке

Процедуры используются для работы с параметрами, передаваемыми программе из командной строки.

Процедура	*Тип*	*Назначение*
CALL GETARG	Подпрограмма	Возвращает *n*-й аргумент командной строки (команда является аргументом с номером 0)
NARGS	Функция	Возвращает общее число аргументов командной строки, включая команду

П.-3.10. Сортировка и поиск в массиве

Процедуры используются для управления хранящимися в массивах данными. Подробно рассмотрены в разд. 6.9.

Процедура	*Тип*	*Назначение*
BSEARCHQQ	Функция	Выполняет двоичный поиск заданного элемента в отсортированном одномерном массиве, содержащем элементы неструктурного типа
CALL SORTQQ	Подпрограмма	Сортирует одномерный массив неструктурного типа

П.-3.11. Управление операциями с плавающей точкой

Процедуры используются для контроля выполнения операций с плавающей точкой и управления реакцией системы на ошибки выполнения.

Процедура	*Тип*	*Назначение*
GETCONTROLFPQQ	Функция	Возвращает значение контрольного слова процессора операций с плавающей точкой
GETSTATUSFPQQ	"	Возвращает значение статусного слова процессора операций с плавающей точкой
CALL LCWRQQ	Подпрограмма	Выполняет то же, что и SETCONTROLFPQQ
SCWRQQ	Функция	Выполняет то же, что и GETCONTROLFPQQ
CALL SETCONTROLFPQQ	Подпрограмма	Устанавливает значение контрольного слова процессора операций с плавающей точкой
SSWRQQ	Функция	Выполняет то же, что и GETSTATUSFPQQ

Литература

1. Бартеньев О. В. Visual Fortran: Новые возможности. - М.: Диалог-МИФИ, 1999. - 288 с.
2. Он же. Графика OpenGL: программирование на Фортране. - М.: Диалог-МИФИ, 2000. - 368 с.
3. Он же. Фортран для профессионалов. Математическая библиотека IMSL: Ч.1. - М.: Диалог-МИФИ, 2000. - 448 с.
4. Он же. Фортран для студентов. - М.: Диалог-МИФИ, 1999. - 400 с.
5. Бродин В. Б., Шагурин И. И. Микропроцессор i 486. Архитектура, программирование, интерфейс. - М.: Диалог-МИФИ, 1993. - 240 с.
6. Вельбицкий И. В. Технология программирования. - Киев: Техника, 1984. - 279 с.
7. Демидович Б. П., Марон И. А. Основы вычислительной математики. - М.: Наука, 1966. - 664 с.
8. Лингер Р., Миллс Х., Уитт Б. Теория и практика структурного программирования. - М.: Мир, 1982. - 408 с.
9. Любимский Э. З., Мартынюк В. В., Трифонов Н. П. Программирование. - М.: Наука, 1980. - 608 с.
10. Майерс Г. Искусство тестирования программ. - М.: Финансы и статистика, 1982. - 176 с.
11. Меткалф М., Рид Дж. Описание языка программирования Фортран 90. - М.: Мир, 1995. - 302 с.
12. Першиков В. И., Савинков В. М. Толковый словарь по информатике. - М.: Финансы и статистика, 1991. - 543 с.
13. Скляров В. А. Язык C++ и объектно-ориентированное программирование. - Минск.: Высш. шк., 1997. - 478 с.
14. Справочник по Автоматизации. - М.: Изд. отд. "Русская Редакция" ТОО Channel Trading Ltd., 1998. - 440 с.
15. Фортран 90. Международный стандарт. - М.: Финансы и статистика, 1998. - 416 с.
16. Холстед М. Начала науки о программах. - М.: Финансы и статистика, 1981. - 128 с.
17. Шикин Е. В., Боресков А. В. Компьютерная графика. Полигональные модели. - М.: Диалог-МИФИ, 2000. - 464 с.

Предметный указатель

E

З

И

K

M

Оглавление