СОВРЕМЕННЫЙ ФОРТРАН.

О.В.Бартеньев

Книга является учебным пособием по языку программирования Microsoft Fortran PowerStation 4.0. В ней содержится полное описание основанных на стандарте Фортран 90 свойств языка, а также рассмотрены средства однооконного и многооконного графического вывода данных.

Книга рассчитана как на начинающих программистов, так и на пользователей со стажем. Для начинающих программистов изложены методы разработки программ и рассмотрены особенности их реализации при программировании на Фортране. Пособие оснащено большим числом примеров, иллюстрирующих рассматриваемые свойства Фортрана. Предназначено для студентов, инженеров и научных работников.

СОДЕРЖАНИЕ

ПРЕДИСЛОВИЕ

В настоящее время, когда существует множество языков программирования, Фортрану всегда отдается предпочтение в научно-технических и инженерных приложениях. В этих областях у Фортрана нет серьезных конкурентов. Это обусловлено тем, что Фортран изначально был создан для научных и численных расчетов и все его последующее развитие ориентировано прежде всего на подобные приложения.

Разработчики Фортран-программ имеют не только современные средства программирования, но и получают доступ к огромному фонду написанного на Фортране программного обеспечения. Созданные на Фортране математические, статистические, графические и иные библиотеки интенсивно используются в различных областях науки и техники.

В книге рассматривается реализация Фортрана - Microsoft Fortran PowerStation версии 4.0, которую для краткости мы будем именовать FPS. Эта разработка основана на стандарте Фортран 90 и позволяет создавать 32-разрядные приложения для Windows 95 и Windows NT. Версия FPS 4.0 вышла сразу вслед за версией FPS 1.0, которая, хотя и была основана на стандарте Фортран 77, включала ряд расширений, продвигающих ее к стандарту Фортран 90, таких, как:

- свободный формат записи исходного кода;
- задание именованных констант;
- новые DO-ЦИКЛЫ и конструкция SELECT CASE;
- возможность использовать в выражениях массивы;
- новые функции обработки строк;
- новые процедуры работы с битами;
- встроенные элементные функции (параметрами и результатом таких функций могут быть согласованные массивы);
- определение производных типов данных;
- динамическое выделение памяти;
- усовершенствованные средства ввода-вывода.

Новая версия совместима с предшествующими версиями Фортрана Microsoft и включает уже все предусмотренные стандартом Фортран 90 свойства. Опишем вкратце некоторые из них.

Теперь FPS поддерживает концепции процедурного и модульного программирования. Иными словами, в программе могут быть наряду с процедурами определены самостоятельные программные единицы - *модули,* которые "скрывают" в себе данные и могут содержать модульные процедуры, выполняющие обработку этих данных. Доступ к не имеющим атрибута PRIVATE данным и процедурам модуля может быть осуществлен в другой программной единице после подключения к ней модуля.

Появились внутренние и рекурсивные процедуры. Вызовы процедур могут выполняться с необязательными и ключевыми параметрами.

Появилась возможность распространять встроенные операции, например сложение или умножение, на производные типы данных (то есть типы данных, которые создаются программистом). Можно также задать новые операции и для встроенных типов данных, например вещественного типа. Это свойство языка называется *перегрузкой операций.* Помимо операций в FPS можно перегрузить присваивание.
Данным как встроенных, так и производных типов можно задать дополнительные свойств. Это выполняется за счет атрибутов, одним из которых является POINTER. Объект данных с атрибутом POINTER называется *ссылкой.* Ссылки являются динамическими объектами и могут быть в процессе выполнения прикреплены к разным *адресатам.* Используя ссылки как компоненты производных типов данных (структур), можно создавать структуры, которые ссылаются сами на себя или другие структуры. Такие структуры чрезвычайно эффективны для создания списков, деревьев и других форм организации данных.

Благодаря новому стандарту Фортран стал более лаконичным. Это достигается в том числе за счет введения в язык конструкций WHERE, *сечений* массивов и большого числа новых, элементных функций и новых встроенных процедур обработки массивов. Теперь там, где ранее требовались циклы, тот же результат достигается одним оператором. Например, для замены всех отрицательных элементов массива *b* нулями достаточно записать оператор *where(b* < 0) *b* = 0. Сумму положительных элементов того же массива вернет функция *sum(b, mask = b > 0).* А чтобы поменять порядок следования элементов вектора c(1:20) на противоположный, достаточно записать c = c(20:1:-1). Столь же компактно можно организовать ввод-вывод массивов и, используя *конструктор массива,* присвоить значения массиву или его сечению.

Улучшены средства для работы с числовыми данными за счет введения большого числа встроенных числовых функций (справочных, элементных, преобразующих) и за счет дополнительных процедур обработки ошибок и управления операциями с плавающей точкой. При работе с действительными и комплексными данными могут быть использованы более **чем** две точности.

Эти и многие другие новые свойств языка существенно ускоряют процесс программирования, позволяют создавать хорошо читаемые, легко изменяемые и быстро работающие программы.

Помимо обусловленных новым стандартом усовершенствований фирма Microsoft наделила FPS свойствами, которые позволяют создавать многооконные графические приложения, диалоговые окна, динамические (DLL) библиотеки, приложения, совместимые с другими платформами и языками программирования. В FPS можно создать и продвинутые Windows приложения, обладающие полным Windows-интерфейсом и имеющие доступ к системным и сетевым функциям и оборудованию. FPS имеет также эффективные средства создания разноязычных приложений, использующих, например, Фортран и СИ++.

В составе FPS вы также обнаружите математическую библиотеку и библиотеку применяемых в статистических исследованиях процедур.

Пособие содержит полное изложение базовых, основанных на стандарте свойств FPS; также в нем подробно, с большим числом поясняющих примеров изложены графические возможности системы.
Характер изложения рассчитан на последовательное освоение языка и методов программирования начинающими пользователями, для которых специально введены первые две главы, содержащие описание основных элементов языка и базовых методов программирования. В последующих главах и приложениях одинаково детально рассмотрены как новые, так и прежние свойства Фортрана.

1. Элементы языка

1.1. Свободный формат 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 *головной*. Она построена по схеме: сначала следует объявление типов используемых переменных, затем - операторы, выполняющие над объявленными переменными некоторые действия. Эта схема является типовой и неоднократно будет воспроизводиться в пособии.

Программа завершается оператором END, в котором *имя-программы p1* и одновременно *program p1* 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 головной программы имен.

Программа записана в *свободном формате*. По умолчанию файл с текстом написанной в свободном формате программы имеет расширение F90. В нем не должно быть метакоманды $NOFREEFORM, а при компиляции нельзя задавать опцию компилятора /4Nf. В свободном формате 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 FPS может быть до 7200 символов. Число строк продолжения при свободном формате 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 свободного программу можно записать и в фиксированном, унаследованном от Фортрана 77 формате (прил. 4). Файлы, содержащие текст в фиксированном формате, по умолчанию имеют расширения F или FOR. В файлах с такими расширениями можно перейти и к свободному формату, задав метакоманду $FREEFORM (прил. 1) или опцию компилятора /4Yf.

Запустим теперь программу `p1`, используя приведенные в разд. 1.2 сведения.

1.2. Консоль-проект

Любая программа рассматривается в FPS как проект. Для запуска новой программы необходимо прежде всего создать проект. В FPS существует несколько типов проектов, однако на первых порах мы будем работать с консоль-проектом - однооконным проектом без графики.

Начнем создание проекта с запуска FPS. Для этого после запуска Windows 95 или Windows NT можно выполнить цепочку действий: Пуск - Программы - Fortran PowerStation 4.0 - Microsoft Developer Studio. Перейдем к созданию нового консоль-проекта. Для этого 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.

Чтобы закрыть проект, следует выполнить: File - Close Workspace.

Существующий проект открывается в результате выполнения цепочки File - Open Workspace - выбрать файл проекта - Open.

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, например: myp.f90 - сохранить.

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

Для удаления файла из открытого проекта достаточно выбрать этот файл в окне File View и нажать Del.

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

Создадим выполняемый EXE-файл: Build - Build.

Запустим созданный EXE-файл: Build - Execute - и получим результат. Для выхода из рабочего окна 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 сочетания клавиш: Ctrl+F8, Shift+F8, F5 или Ctrl+F5.

Сохраняемые на диске файлы с исходным текстом программы могут иметь расширение F90, F и FOR. Например, myp.f90, myp.f, myp.for. В первом случае компилятор считает, что файл написан по принятым в

Фортране 90 правилам, то есть в свободном формате. В двух последних по умолчанию предполагается, что исходный текст записан в фиксированном формате (прил. 4). Мы же будем использовать для файлов расширение F90 и свободный формат 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 выполнены 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, то есть 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 всегда 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 разде-

ляющее 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. Следующий вывод эквивалентен предыдущему
print *, a(1), a(2), a(3)
end
```

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 головной программы или встроенной процедуры, и локальные, например имя переменной или константы.

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

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
```

Операндами выражения могут быть константы, простые и составные переменные, подобъекты составных переменных 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.

В зависимости от типа операндов 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 FPS приведены в табл. 1.1.

Таблица 1.1. Встроенные арифметические операции

Действие	*Обозначение*	*Пример*	*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, st2*3 /'C6'/        ! Объявление символьных переменных
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, то есть 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. То есть выражение $d = (a + b)/2$ эквивалентно

```
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 равнство (=) и записывается в виде:

varname = выражение

В результате присваивания переменная *varname* получает новое значение, которое получается 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 к типу *varname*. Поскольку в результате преобразований возможна потеря точности, то необходимо следить, чтобы эта потеря не привела к 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 выражения любого типа и вида (арифметические, логические, константные...); список ввода - только переменные.

Последняя или единственная звездочка 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 такой ввод:

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

Результат вывода:

```
1.00000      2.000000      3.000000
```

Замечание. В качестве разделителя задаваемых на клавиатуре значений можно использовать и запятые или запятые вместе с пробелами, например:

```
1, 2, 3
```

1.8.1. Некоторые правила ввода

Для рассмотрения правил ввода введем ряд понятий.

Запись 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.

Ввод под управлением списка выполняется по правилам:

- поля записи могут разделяться пробелами и запятой;
- если между полями записи присутствует слэш (/), то ввод прекращается;
- каждый оператор ввода (если не задана опция ADVANCE = 'NO') выполяет 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 (то есть если выполняется попытка чтения записи "конец файла" и за пределами файла);
- Если размещенные на читаемом поле символы не могут быть преобразованы 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              ! Ввод 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 самостоя-

тельно, 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 прочитаны несколько (или все) страницы первой 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 типа*:

```
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 вам понять работу изучаемых элементов языка, убедиться в правильности работы программы 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 текстовый файл могут быть занесены с клавиатуры в среде FPS так же, как и в другом текстовом редакторе. В FPS для создания нового файла используйте последовательность File - New - Text File - OK. Далее вве-

дите 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 5 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
enddo
```

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
enddo
```

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 среде FPS, 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 иллюстрирующих изучаемый материал 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 в FPS имеют расширения F90, FOR или F, например koda.f90. По умолчанию FPS считает, что файлы с расширением F90 написаны в свободном формате, а с расширениями FOR и F - в фиксированном.

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 модулей. Программа, преобразовывающая исходный код в объектный, называется *компилятором* или *трансля-*

тором. Файлы с объектным кодом - объектные модули - имеют расширение OBJ, например koda.obj.

На следующей стадии обработки выполняется размещение объектных модулей программы в памяти компьютера. Часть объектных модулей может быть загружена из библиотек. При этом отдельные компоненты (головная программа, модули, подпрограммы, функции) связываются друг с другом, в результате чего образуется готовая к выполнению программа - *исполняемый файл*. Расширение таких файлов EXE. На этапе генерации исполняемого кода также могут возникать ошибки, например вызов несуществующей подпрограммы.

В 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 среде FPS оказываются доступными две реализации: Debug и Release. В реализации Debug активизирован отладочный режим. Перечень выполняемых компилятором в этом режиме проверок приведен в прил. П.1.4.2. Ниже приведены задаваемые по умолчанию 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/fpt2.pdb"
```

Опции компоновщика:

```
kernel32.lib /nologo /subsystem:console /incremental:yes
/pdb:"Debug/fpt2.pdb" /debug /machine:I386 /out:"Debug/fpt2.exe"
```

Реализация Release.

Опции компилятора:

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

Опции компоновщика:

```
kernel32.lib /nologo /subsystem:console /incremental:no
/pdb:"Release/fpt2.pdb" /machine:I386 /out:"Release/fpt2.exe"
```

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 $y = x*sinx$ 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 ≤ b выполнять:
      вычислить значение функции y в точке x: y = x*sinx
      вывести значения x и y
      перейти к следующей точке отрезка: x = x + dx
    конец цикла
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
enddo                           ! 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 ... ENDDO следует использовать правило *рельефа*, состоящее в том, что операторы DO WHILE и ENDDO, 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 ENDDO. Еще одно полезное правило: после запятой или иного разделителя в тексте программы следует проставлять *пробел*, то есть 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              ! Объявление имен и типов переменных
 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
 enddo
end program txy                      ! Завершаем программу txy
```

Замечание. FPS обладает специальной библиотекой процедур DIALOGM, предназначенных для создания диалоговых окон В/В данных.

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, то есть 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 в выводимой записи. При форматном выводе это необходимо, поскольку первый символ каждой записи не печатается и рассматривается как символ управления кареткой.

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 опцию 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 (то есть 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 ... ENDDO.

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 так далее.

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.

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. Размещенные между DO и END DO операторы и конструкции образуют *тело цикла*.

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).

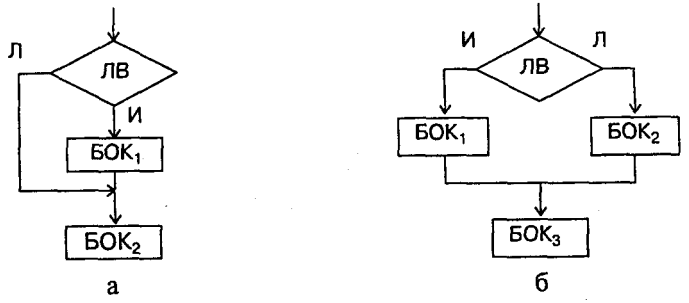

Рис. 2.1. Ветвления: а - ветвление "если - то"; б - ветвление "если - то - иначе"

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 (заключенные в квадратные скобки элементы линейной схемы могут быть опущены):

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

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
```

Запись ветвления "если - то - иначе" в FPS:

```
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 обратить внимание на то, 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 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. Схема решения:

```
1°. Начало
2°. Найти наименьшее из трех чисел
    и присвоить результат переменной m3
3°. Если ma равно m3, то вывести сообщение "Число ma"
4°. Если mb равно m3, то вывести сообщение "Число mb"
5°. Если mc равно m3, 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, d
 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
```

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 в FPS может иметь целый или вещественный тип. Параметры p_s, p_e и шаг s являются выражениями целого или вещественного типа.

Графически цикл "с параметром" иллюстрирует рис. 2.2.

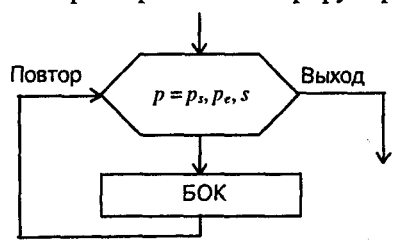

Рис. 2.2. Цикл с параметром

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 ≥ pe, 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 исполняемый код при компиляции программы.

2. В FPS в цикле "с параметром" запрещается в теле цикла менять значения переменной цикла *p*. Изменение параметров в 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 = ps, pe, s [ выполнить: ]
      БОК
    конец цикла [ с парметром p]
```

Наиболее часто в FPS цикл с параметром записывается так:

```
DO p = ps, pe [, 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*, 4*a*, ..., n^2a.

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

```
L = L + i² * a  (i = 1, 2, ..., n),    (*)
```

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. Схема решения:

```
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
5°. Вывод L
6°. Конец
```

```
program polen
 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
 enddo
 write(*, *) ' L = ', L
end
```

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
enddo
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 FPS 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.

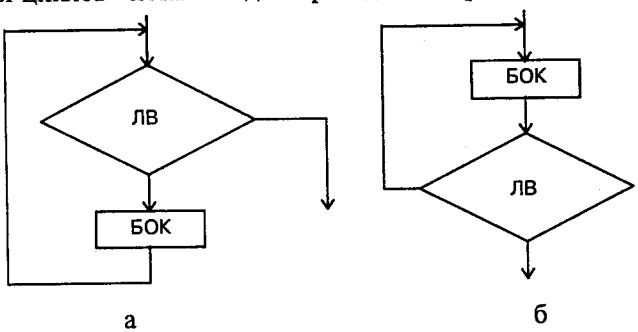

Рис. 2.3. Циклы "пока" и "до": а - цикл "пока"; б - цикл "до"

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 в FPS:

```
Цикл "пока":
X°. Пока истинно ЛВ  [выполнять:]
        БОК
    конец цикла
DO WHILE ( ЛВ )
  БОК
END DO
```

```
Цикл "до":
X°. Выполнять:
        БОК
    если ложно ЛВ, то выход
      из цикла
  конец цикла
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 следует 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 их цикла
  else
   cycle                             ! Переход на начало цикла
  endif
 endif                               ! pos - число положительных элементов массива
 if (va > 0) pos = pos + 1
enddo                                ! 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". Схема алгоритма:

```
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 ≤ n и ze ≤ k выполнять:
 va = a(i)
 Если va ≠ 0, то
  ze = ze + 1
 иначе, если va > 0, то
  pos = pos + 1
 конец если
  конец цикла
5°. neg = n-ze-pos                   ! neg - число отрицательных элементов массива a
```

```
6°. Вывод pos и neg
7°. Конец
program pnw
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
 endif
 i = i + 1
enddo                          ! 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
enddo
```

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 следует 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 он 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, по-

ка 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 FPS для реализации фрагмента можно использовать программные компоненты: *головную 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 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.

Линейная схема алгоритма:

```
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.
        ! Схема алгоритма этого фрагмента приведена в разд. 2.2.2.
    конец если
6°. Конец
```

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

Интерфейс к фрагменту 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.

Схема алгоритма 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
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 имя (если не задано предложение RESULT) и тип совпадают с именем 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'
endif
end

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)
 enddo
 if (md > 0) md = 0                ! или: md = min(md, 0)
end function
```

2.3.2. Использование подпрограмм

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

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
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 головной 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
  endif
 end function
end module

program txy                           ! Заголовок головной программы
use ched                              ! Подключаем модуль ched
real x, y                             ! Объявление данных головной программы
if (moched( )) then                   ! Запуск 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
 enddo
endif
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 ею 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, то есть способности алгоритма получать требуемые результаты, и эффективности - способности алгоритма получать нужные результаты за приемлемое время.

Параллельно с разработкой алгоритмов решаются вопросы *организации данных*, то есть 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 ней (нем) ошибок. *Отладка* - процесс локализации и исправления ошибок. В результате тестирования устанавливается, соответствуют или нет разработанные фрагменты и состоящая из них программа сформулированым в спецификациях требованиям. Методы тестирования и отладки рассмотрены, например, в [5].

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 име-

ют 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 цели устранения выявленных пользователями ошибок и адаптации 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. Например, имя *g* может быть использовано для обозначения ускорения свободного падения. Понятен смысл и имени *pi*.

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

```
integer, parameter :: n = 20, m = 10! Размерности матрицы аа 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.
- Внутренняя функция лучше операторной.
- Если формальным параметром процедуры является массив, то его следует оформлять как массив, перенимающий форму или заданной формы.
- Внутренние массивы и строки процедур следует оформлять как автоматические объекты.
- Не использовать оператор GOTO.
- Не применять избегать ассоциирования памяти.
- Открытые от использования нерекомендуемых и устаревших средств Фортрана (прил. 4).

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 объектом данных является и *функция*, поскольку она так же, как обычные переменные и константы, обладает типом и используется в качестве операнда выражения.

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 90 каждый встроенный тип данных характеризуется параметром разновидности (KIND). Для числовых типов данных этот параметр описывает точность и диапазон изменения. Для символьного типа данных в FPS 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 FPS. 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 по 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)

Типы	*Разновидность*	*Число байт*	*Примечание*
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 в FPS можно использовать при задании типов данных синтаксис Фортрана 77.

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

FPS включает большое число встроенных числовых справочных и преобразующих 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 метакоманду !MS$INTEGER:2.

В FPS синтаксис оператора INTEGER таков:

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 атрибут, описывающий представленные в *entity-list* объекты данных. Если хотя бы один атрибут указан, то должен быть использован разделитель :: . Возможные атрибуты: ALLOCATABLE, DIMENSION(dim), EXTERNAL, INTENT, INTRINSIC, OPTIONAL, PRIVATE, PUBLIC, PARAMETER, POINTER, SAVE и TARGET. 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.

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 метакоманда !MS$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 массивов
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'
 endif
enddo
```

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 метакоманда $STRICT, при которой все расширения FPS по отношению к стандарту Фортран 90 воспринимаются как ошибки.

3.2.2. Объявление данных вещественного типа

Синтаксис оператора объявления объектов вещественного типа в FPS 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 метакоманды !MS$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) :: testval = 50._8
real x, a(10), b(20, 30)
optional testval                    ! Теперь зададим атрибуты
save a, b
parameter (x = 100.0)
```

3.2.3. Объявление данных комплексного типа

Коплексное 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)
```

Первое число пары представляет действительную, а второе - 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

В FPS 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 метакоманду !MS$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 опции 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 метакоманду !MS$INTEGER:2 и для вещественного типа - опцию компилятора /4R8 или метакоманду !MS$REAL:8 (прил. 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, а объекты, имена которых начинаются с букв *n* и *N*, по умолчанию будут иметь тип CHARACTER (*len* = 10).

Задание одной и той же буквы в операторе (непосредственно или через диапазон) недопустимо. Диапазон букв должен быть задан в алфавит-

ном 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                    ! integer
name = 'Peter'               ! character(10)
c%e  = 1                     ! целочисленный компонент типа feg
$j   = 5.                    ! 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, для задания значений именованных констант, начальных значений переменных и в качестве фактических параметров подпрограмм и функций. 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, восмеричной 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, восмеричной 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'/              ! шестадцатеричное 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 могут быть опущены, 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* и *часть* - целая или вещественная константы.

Буквальные комплексные константы могут быть одинарной (COMPLEX(4) и COMPLEX) и двойной (COMPLEX(8) и DOUBLE COMPLEX) точности. В комплексных константах одинарной точности мнимая и действительная части занимают по 4 байта памяти; в комплексных константах двойной точности - по 8 байт памяти. 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 символа 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 дины - 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 должн 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.

В FPS 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 восмеричном или шестнадцатеричном коде, которые указываются при задании констант после обратной наклонной черты. Например, '\62'c и '\x32'c задают символ '2' в восмеричном и шестнадцатеричном кодах (ASCII код символа '2' равен 50). Также в СИ существует специальная запись приведенных в табл. 3.2 часто используемых символов, называемых эскейппоследовательностями.

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 хвостовые пробелы и *null*-символ.

Таблица 3.2. Эскейппоследовательности

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		Восмеричная константа
\xhh		Шестнадцатеричная константа

В восмеричном 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
```

В Фортране, начиная с версии FPS4, может быть задана символьная константа нулевой длины.

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

В FPS1 для задания такой константы использовалась СИ-строка:

```
character ch /''c/
```

Если в СИ-строке после обратного слэша (\) указан символ, отсутствующий в табл. 3.2, то слэш будет проигнорирован, например:

```
character(8) st1 /'\x\y'c/
character(8) st2 /'\x\y'/
write(*, *) st1,  st2           ! xy   \x\y
```

Поскольку символьные строки завершаются *null*-символом, то при их конкатенации 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, то есть 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. В FPS 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.

В FPS 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 )
complex(4) z                        ! Сначала объявляется тип, а
parameter ( z = -(12.45, 6.784) )   ! затем задается значение
write (*, *) bell                   ! Звуковой сигнал
write (*, *)  (bell, i = 1, 10)     ! Продолжительный сигнал
```

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 соот-

ветствующей 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 хвостовых 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, начиная с версии FPS4, 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 *st3* 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. FPS поддерживает одно значение параметра разновидности (KIND = 1) для символьных объектов данных.

attrs - один или более атрибут, разделенные запятыми. Если хотя бы один атрибут задан, наличие разделителя :: обязательно. Возможные атрибуты: ALLOCATABLE, DIMENSION(dim), EXTERNAL, INTENT, INTRINSIC, OPTIONAL, PARAMETER, POINTER, PRIVATE, PUBLIC, SAVE и TARGET. Если задан атрибут PARAMETER, то необходимо также задать 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 /' '/                ! Строка 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 хвостовыми. Если начальное значение содержит больше символов, чем длина строки, то избыточные символы отсекаются.

В FPS могут быть использованы символьные буквальные СИ-константы, завершаемые *null* (CHAR(0)) символами, например:

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

Как и для других типов данных, в FPS 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. 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*'??? '.

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(*) 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
 print *, len(st3)                   ! параметра
...
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
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

Символьные 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. Такие 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).

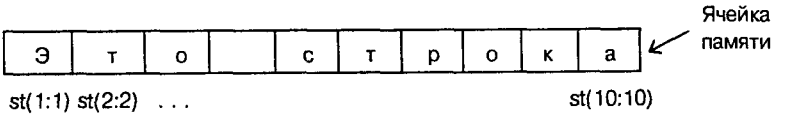

Рис. 3.1. Строка в памяти ЭВМ

Каждый символ строки имеет номер. Причем 1-й символ имеет номер 1, 2-й - номер 2 и так далее. 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 хвостовых
```

```
enddo                          ! пробелов
print *, ' k =', k             ! 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 единицы, 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 хвостовых пробелов.

3. Функция TRIM возвращает строку без хвостовых пробелов.

4. В FPS для выделения строки без хвостовых пробелов лучше применить функцию TRIM, например:

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

То же самое в FPS1 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, то есть 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 чис-

лу 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 - номер устройства. При ссылке на внутренний файл номером устройства является имя 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 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 с символьными данными. Встроенные функции FPS разделяются на элементные, справочные и преобразующие. 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 метакоманда !MS$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. В России принято, что прописная буква 'А' русского алфавита имеет код 128, то есть функция ICHAR('А') возвращает число 128, и наоборот, CHAR(128) есть первая буква русского алфавита 'А'.

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
 endif
enddo
if (n == 0) print *, 'Нет различий'
end
```

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

Замечание. Функции IACHAR, ICHAR, ACHAR, CHAR являются элементными, то есть 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 выполняется сравнение кодов (ASCII) сравниваемых 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
 endif                                    ! Новиков.
enddo                                     ! 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 хвостовых 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 хвостовых пробелов и их последующей вставки в начало строки, например:

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

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, то есть 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 положения символа строки *set* в строке *string*. 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.

Пример.

```
integer  array(2)
print *, scan ('Fortran', 'tr')                !      3
print *, scan ('Fortran', 'tr', back = .true.) !      5
print *, scan ('Fortran', 'gha')               !      6
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 хвостовые 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* присутствуют 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', 'nbca')         !      0
```

Замечания:

1. В FPS4 дополнительно включены функции: IACHAR, ACHAR, ADJUSTL, ADJUSTR, REPEAT и TRIM. Все остальные символьные функции были доступны в Фортране 77.

2. В FPS 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 символы ,, ., ;, :, -, !, ?, табуляции, а также сочетания названных символов с пробелами.

Поясним содержание задачи примером. Пусть текстовый 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 хвостовых 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
   endif
  enddo
! После просмотра всей строки и, если строка не была пустой,
! мы обязаны добавить последнее слово в массив words
  if (wb > 0) call addword(words, st, wb, we, nw)
enddo
close(unit)
write(*, *) 'Число слов в тексте nw =', nw
do j = 1, nw
  write(*, *) words(j)
enddo
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 так далее. 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. В FPS записи вводятся оператором TYPE или, как в FPS1, 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, включая их в текст 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. Входящие в *component decl* операторы не могут содержать начальные значения переменных.

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
```

После введения производного типа данных объекты (переменные или константы) нового типа вводятся 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 метакоманды !MS$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 функции
  function mu(a, b)                  ! необходимо 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 проце-

дуре 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                    ! Головная программа
 type point                     ! В головной 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 последующим *use*-включением модуля в программные единицы, использующие 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 головной программы
end subroutine
```

3.9.5. Запись как результат функции

В FPS 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 его 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)                   ! Наполням 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
enddo
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
 enddo
 contains
 integer function rmark()           ! Генератор экзаменационных оценок
  real(4) rnd
  call random(rnd)                  ! rnd - случайное число real(4) от 0. до 1.
  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. Начиная с версии FPS4 структурная переменная может быть записана "целиком" как в неформатный (двоичный), так и в текстовый файл (в более ранних версиях Фортрана в текстовом файле запись можно было сохранить лишь поэлементно). При передаче записи *stud* в текстовый

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 1 Оценка 2 Оценка 3 Оценка 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 *'Старт 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 головной 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 FPS1

3.9.7.1. Объявление и присваивание значений

FPS4 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 метакомандой $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 (допустимо в FPS4)
```

```
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.

В FPS 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) следует вводить оператором TYPE ... END TYPE, используя затем для объявления записи (перемен-

ной 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 текстовый (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 без применения USE модуля 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 FPS используются целочисленные указатели.

Целочисленный указатель - это переменная, содержащая адрес некоторой ячейки памяти и связанная с некоторой переменной, называемой *адресной переменной*. Целочисленный указатель имеет три 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
enddo                              ! 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
enddo                         ! 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 следит 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 функцией 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)
p0 = malloc(size)                  ! Функция malloc возвращает начальный адрес
                                   ! выделенной памяти
p = p0                             ! Установим указатель p в начало выделенной памяти
do k = 0, size
 gk = int(k, kind = 1)             ! В ячейку с адресом p заносится значение gk
 p = p + 1                         ! Переход к следующему байту памяти
enddo                              ! Просмотр памяти
print '(10i3)', (gk, p = p0, p0 + 5)! 1  2  3  4  5  6
call free(p0)                      ! Функция free освобождает выделенную функцией
end                                ! malloc память
```

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

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, во-первых, прикрепленная ссылка, во-вторых, переменная, имеющая атрибут TARGET (объявленная с атрибутом TARGET или в операторе с тем же именем), и, в-третьих, 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 а присоединена к d2
d => d2; e => d2                    ! Несколько ссылок присоединены к одному адресату
print *, a, d, e                    !      99       99       99
a = 100                             ! Изменение а вызовет изменение всех
                                    ! ассоциированных с а 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 к неразмещенному
адресату
allocate(b2(5))
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) /)
a => b(3:10:3)                    ! нижняя граница массива a равнаединице, верхняя -
трем
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 возвращает .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. 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.

3.11.4. 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*. Последний элемент списка содержит нулевой указатель. Однонаправленный список с записями одинаковой длины можно представить в виде рис. 3.2.

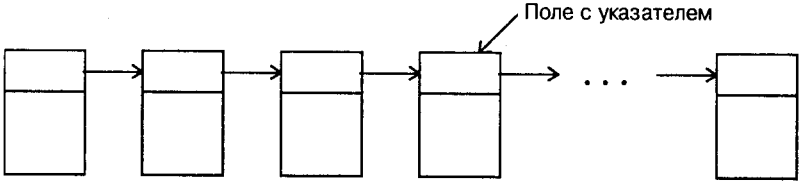

Рис. 3.2. Схема однонаправленного списка с записями одинаковой длины

Запишем текст программы формирования и редактирования однонаправленного списка из 10 элементов. Пусть помимо поля с указателем каждый элемент списка содержит еще два поля, одно из которых (index) используется для индексирования элементов списка.

```
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
interface                             ! Процедуры, имеющие формальные
 subroutine wali(top)                 ! параметры-ссылки, должны обладать
  use eni                             ! явно заданным интерфейсом
  type (entry), pointer :: top
 end subroutine wali
end interface
type (entry), pointer :: tree, top, current, newtop
integer i
nullify(tree)                         ! Обнуляем конец списка. В конце списка
                                      ! associated(tree) будет возвращать .false.
do i = 1, 10                          ! Цикл формирования списка
 allocate (top)                       ! Размещаем новый элемент списка и
 top = entry(11.0*i, i, tree)         ! заносим информацию в поля
                                      ! с данными
                                      ! Ссылка top содержит в качестве
                                      ! компонента ссылку tree
 tree => top                          ! Установим указатель списка на вновь
enddo                                 ! добавленный элемент
! Список сформирован. top ссылается на последний элемент списка
call wali(top)                        ! Просмотр списка
```

```
 tree => top                          ! Перемещение в вершину списка
 do while(associated(tree))           ! Изменим теперь значение поля val у
  if(tree%index == 5) then            ! элементов списка с index = 5 и index = 4
    tree%val = 500                    ! Поле val элемента списка с index = 5
    tree%next%val = 400               ! Поле val элемента списка с index = 4
    exit
  endif
  current => tree%next
  tree => current
 enddo
 call wali(top)                       ! Просмотр списка
 tree => top                          ! Исключим из списка элемент с index = 7
 do while(associated(tree))
  if(tree%index == 8) 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
   endif                              ! элемента из списка
   exit
  endif
  current => tree%next
  tree => current
 enddo
 call wali(top)                       ! Просмотр списка
 tree => top                          ! Вставим элемент после элемента с index=8
 do while(associated(tree))
  if(tree%index == 8) then
   allocate (newtop)                  ! Вставка элемента
   newtop = entry(700, 7, tree%next)
   tree%next => newtop
   if (tree%index > 1) then
    newtop%next => tree%next%next
   else                               ! При вставке после последнего элемента
    nullify(newtop%next)              ! обнуляем ссылку
   endif
   exit
  endif
  current => tree%next
  tree => current
 enddo
 call wali(top)                       ! Просмотр списка
 end program one_d_li
 subroutine wali(top)                 ! Подпрограмма просмотра списка начиная
  use eni                             ! с элемента top
  type (entry), pointer :: top, tree, current
  tree => top                         ! Присоединяем ссылку tree к элементу top
  do while(associated(tree))          ! Цикл просмотра списка
   print *, tree%val, tree%index
   current => tree%next
   tree => current
  enddo
  read *                              ! Ожидаем нажатия Enter
 end subroutine wali
```

Замечание. Используя ссылки в качестве компонентов производного типа, можно сформировать циклические, двунаправленные списки и более сложные структуры данных.

3.11.5. Ссылки как параметры процедур

Ссылка может быть формальным параметром процедуры. В этом случае фактический параметр также должен быть ссылкой. Ранги фактического и формального параметром - 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 атрибута 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  2.0  3.0  4.0  5.0
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
```

3.11.6. 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).

3.11.7. 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 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. В FPS, 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.

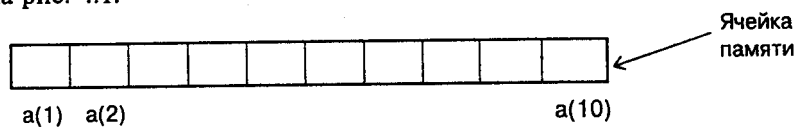

Рис. 4.1. Расположение элементов массива в памяти ЭВМ

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 встроенный DO-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)
enddo
write(*, *) ' s = ', s                ! s = 47.7
! В FPS для решения данной задачи достаточно применить функцию 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.

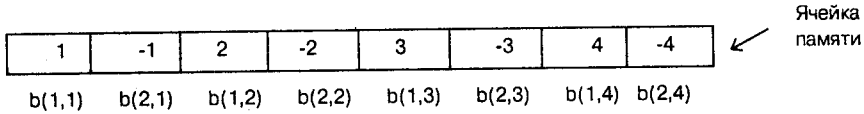

Рис. 4.3. Расположение элементов двумерного массива в памяти ЭВМ

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, то есть 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)
  enddo
 enddo
enddo
```

Иными словами, при размещении многомерного массива в памяти ЭВМ быстрее всего изменяется самый левый индекс массива, затем следующий за ним индекс и так далее.

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 i = -2, 1
 do j = 6, 8
  if (b(i, j) > 0) p = p * b(i, j)
 enddo
enddo
write(*, *) ' p = ', p                  ! p = 193.261200
! В FPS для вычисления произведения можно применить функцию PRODUCT:
```

```
p = product(b, mask = b > 0.0)
write(*, *) ' p = ', p                    ! p = 193.261200
```

Индексом массива может быть целочисленное выражение:

```
real :: b(5, 10) = 5.1
real :: a(5, 5), c(30), r = 7
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 размерности массива DIMENSION могут быть использованы атрибуты INTENT, OPTIONAL, PARAMETER, PRIVATE, PUBLIC, SAVE и TARGET.

4.2. Массивы нулевой длины

В FPS начиная с версии 4.0 массив, как, впрочем, и строка, может иметь нулевую длину. 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 нулевой длины 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 (2, 5). 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

В FPS можно получить доступ 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 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 равной нижней (верхней) границе 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
enddo                                ! 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
enddo
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) получат 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) получат 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 использовано и в качестве параметра встроенных функций обработки массивов, элементных и справочных функций.

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 (см. 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 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 массивам значений

Как было показано выше, массив может быть определен при инициализации в операторах объявления типа или в операторе DATA. Также изменить значения элементов массива можно в результате выполнения оператора присваивания, 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 слэшем *не допускаются.*

Список-значений может содержать последовательность скаляров, встроенных DO-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.

Встроенный DO-*цикл* имеет вид:

(выражение | встроенный DO-*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.

Встроенный DO-цикл добавляет в список значений

max(stop - start + inc / inc, 0)

элементов. Выражение может содержать *dovar.* Возможна организация вложенных встроенных DO-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 - встроенные DO-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 встроенные DO-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 циклического 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. Оператор и конструкции WHERE

В FPS 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)
enddo
```

Синтаксис оператора WHERE:

WHERE (*логическое выражение - массив*) *присваивание массива*

Синтаксис конструкций 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*. Результатом его 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 присваиваний 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
endwhere
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, то есть 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
enddo
```

Нельзя передавать управление в тело конструкции WHERE, например, посредством оператора GOTO.

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 FPS 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 a2[allocatable](:)                  ! Атрибут allocatable в FPS1
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. *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
endif
```

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)))          ! Неправильно
```

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 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                         ! Размещение work не требуется, поскольку
end subroutine delo                     ! оно уже выполнено в подпрограмме rewo
```

Таким же свойством обладают и ссылки. То есть если в последнем примере в модуле *wara* объявить ссылку

```
integer, pointer :: work(:,:)
```

то выделенный ей в подпрограмме *rewo* адресат будет доступен и в головной 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 открепление 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
```

Замечание. Хорошей практикой является освобождение оператором DEALLOCATE выделенной динамической памяти, когда надобность в ней отпадает. Это позволяет избежать накопления неиспользуемой и недоступной памяти.

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 поцедуры 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 с *заданной формой.*

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
enddo                                 ! 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
  endif
 enddo
 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 ранг и форма фактического и формального параметров совпадают. 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.

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)
```

То есть 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 (то есть 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

FPS 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 ко-

торого 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)                   !    4.8400...       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
   endif
  enddo
  return
 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 работы с массивами

В FPS 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 из приводимых ниже функций 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 метакоманда !MS$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*(2, 3) по второму измерению 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, что и для функции ALL.

Пример для функций ALL, ANY, COUNT:

```
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
```

MAXLOC(*array* [, *mask*]) - возвращает одномерный массив стандартного целого типа. Размер массива-результата равен рангу массива *array*. Значения элементов массива-результата равны индексам максимального элемента целочисленного или вещественного массива *array*. Значение максимального элемента удовлетворяет заданным (необязательным) условиям *mask*. Если несколько элементов содержат максимальное значение, то берется первый по порядку их следования в *array*.

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*.

MINLOC(*array* [, *mask*]) - возвращает одномерный массив стандартного целого типа. Размер массива-результата равен рангу массива *array*. Значения элементов массива-результата равны индексам минимального элемента целочисленного или вещественного массива *array*. Значение минимального элемента удовлетворяет заданным (необязательным) условиям *mask*. Если несколько элементов содержат минимальное значение, то берется первый по порядку их следования в *array*.

Смысл массива *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*.

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 = 3)
```

```
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
```

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
print *, minval(array, dim = 1)        !    1    2    3
print *, minval(array, dim = 2)        !    1    2
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 те 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 те 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 сумму произведений элементов с равными значениями индексов одномерных массивов (векторов).

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(CONJ(*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 математике правилам умножение матриц целого, вещественного, комплексного и логического типа.

matrix_a - одномерный или двумерный массив целого, вещественного, комплексного или логического типа.

matrix_b - 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
```

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.

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 нижние 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. *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 выходной 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 так далее. 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
```

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*. *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)      ! Результат: 1 8 3
                                   !            8 2 8
```

4.12.4.3. Переформирование массива

RESHAPE(*source*, *shape* [, *pad*] [, *order*]) - формирует массив заданной формы *shape* из элементов массива *source*. Результирующий массив имеет те 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 - индекс *order*(n). При этом элементы из *source* выбираются в нормальном порядке. Далее при нехватке элементов *source* следуют копии элементов *pad*. Параметр *order* позволяет в частно-

сти 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 /))
! Результат: 1  2  3  4  5
!            6  0  0  0  0
! 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. $1 \le dim \le n$, 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
ar1 = cshift(array, 1, dim = 1)
! Сдвиг в каждом столбце на одну позицию
! Результат: 4 5 6
!            7 8 9
!            1 2 3
ar2=cshift(array, shift=(/-1, 1, 0/), dim = 2)
! Сдвиг в первом ряду на -1, во втором - на 1
! 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
endif
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
endif                                  ! 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 слэша (/), 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 = check) a(i)
enddo
```

нам потребовалось бы расположить в текстовом файле каждое число на отдельной строке, то есть 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 = check) 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 выполняется так:

```
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)
enddo
```

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 (то есть 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)
enddo
```

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 (разд. 10.5).

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 другие. 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, то есть 1 и -1. Затем, поскольку переменные *dp* и *dn* имеют тип REAL(4), целые числа 1 и -1 будут преобразованы к стандартному вещественному типу.

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 целочисленого 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 разных 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 *d2* 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 к типу 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)                  ! Правильно
```

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. То есть 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
enddo
```

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
enddo
```

Общий вывод из приведенного примера: нельзя сравнивать вещественные числа на предмет точного равенства или неравенства, а следует выполнять их с некоторым приближением.

Не следует комбинировать переменные, различия между которыми превышает число значащих цифр, например:

```
real(4) :: x =  1.0e+30, y = -1.0e+30, z =  5.0
print *, (x + y) + z                !     5.000000          (правильно)
print *, x + (y + z)                !     0.000000E+00      (ошибка)
```

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 операций 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 в FPS могут быть записаны в двух формах:

.LT. или < меньше;

.LE. или <= меньше равно;

.GT. или > больше;

.GE. или >= больше 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 к вещественному типу.

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 хвостовых пробелов, например выражение 'Expression' > 'Exp1' вычисляется как 'Expression' > 'Exp1   '.

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 FPS также допускает 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 к целому типу 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
```

Операции отношения и логические операции выполняются слева направо, то есть, 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).

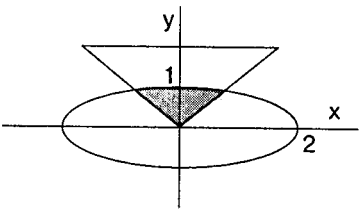

Рис. 5.1. Условие попадания точки в область

```
if(x**2/4 + y**2 < 1.0 .and. y > abs(x)) then
  write(*, *)  'Inside'
else
  write(*, *)  'Outside'
endif
```

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 FPS, позволяющие смешивать логические и целочисленные данные, являются расширением FPS 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 перегружены 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, то есть 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 .с. 10 приведет к ошибке, поскольку типы операндов отличны от REAL(4).

При перегрузке операций отношения, для обозначения которых в FPS 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 операций 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

**	*	+	//	.LT., <	.EQ., ==	.NOT.	.AND.	.OR.	.XOR.
	/	-		.LE., <=	.NE., /=				.EQV.
				.GT., >					.NEQV.
				.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 встроенного DO-цикла;
- конструктор структуры, компоненты которого являются константными выражениями;
- обращение к встроенной элементной или преобразующей 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 данных 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 преобразующих 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 использоваться скалярные, неконстантные выражения. (Наряду, разумеется, с константными.) 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 не должны объявляться ранее их использования в выражении, за исклю-

чением 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
 real(kind(x)) y, fun
 integer lob(2)
 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 в FPS 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, FPS имеет 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*. То есть присваивание *x* = *y* эквивалентно вызову *call sub*(*x*, *y*).

Пример. Присвоить целочисленной переменной сумму кодов символов строки.

```
interface assignment ( = )
 subroutine chati ( n, ch )
  integer, intent (out) :: n
  character(*), intent (in) :: ch
 end subroutine
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))
 enddo
end subroutine
```

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:

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.

Справочные функции выдают информацию о свойствах параметров функций. Результат справочной функции не зависит от значения параметра, который, в частности, может быть и неопределенным.

Преобразующие функции.

Неэлементные подпрограммы.

Помимо встроенных FPS имеет большое число дополнительных процедур, перечень которых приведен в прил. 3.

6.2. Обращение с ключевыми словами

Поскольку встроенные процедуры обладают явно заданным интерфейсом, вызов большинства процедур может быть выполнен с ключевыми словами. Например:

```
pi = asin(x = 1.0)
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 необязательными ключевыми словами (порядок не нарушен)
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 MAXVAL 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-spec* 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, к какому допустимому типу (вещественному или комплексному) 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 метакоманда !MS$INTEGER:2, то возвращаемый функцией результат имеет тип INTEGER(2). При этом также меняется устанавливаемый по умолчанию тип логических данных, то есть 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 метакоманда !MS$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* к целому типу 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 к целому типу 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	*Тип параметра*	*Тип результата*
INT1(*a*)	Целый, вещественный или комплексный	INTEGER(1)
INT2(*a*)	" " " "	INTEGER(2)
INT4(*a*)	" " " "	INTEGER(4)
HFIX(*a*)	" " " "	INTEGER(2)
JFIX(*a*)	" " " "	INTEGER(4)

IZEXT(*a*), JZEXT(*a*) и ZEXT(*a*) - преобразовывают логические и целые значения к целому типу с большим значением параметра разновидности. Преобразование выполняется путем добавление нулей в свежие биты результата.

Функция	*Тип параметра*	*Тип результата*
IZEXT(*a*)	LOGICAL(1), LOGICAL(2),	INTEGER(2) BYTE, INTEGER(1), 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* к вещественному типу 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* к вещественному типу REAL(8).

CMPLX(*x* [, *y*] [, *kind*]) - преобразовывает целые, вещественные или комплексные параметры к комплексному типу 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. То есть значение отдельных битов результата и *source* совпадают. Тип и параметры типа результата такие же, как у *mold*.

Пусть физическое представление *source* есть последовательность *n* битов $b_1b_2...b_n$, а представление *mold* занимает *m* бит, тогда результат:

при $n = m$ $b_1b_2...b_n$;

при $n < m$ $b_1b_2...b_ns_1s_2...s_{m-n}$, где биты s_i не определены;

при $n > m$ $b_1b_2...b_m$.

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*) - возвращает наименьшее стандартное целое, большее или равное значению вещественного аргумента *a*.

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*) - возвращает наибольшее стандартное целое, меньшее или равное значению вещественного аргумента *a*.

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/))     ! Возвращает   (8, -6)
```

MOD(*a*, *p*) - возвращает остаток от деления *a* на *p*, то есть 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 $b \geq 0$, и -1, если $b < 0$. Параметры *a* и *b* должны быть либо оба целыми, либо оба вещественными.

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.

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 ($-\pi < \vartheta \leq \pi$).

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. То есть, 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*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 FPS существуют тригонометрические функции, в которых аргумент должет 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 $-\pi/2 \le$ ASIN(x) $\le \pi/2$.

ASIND(x) - возвращает арксинус вещественного аргумента x ($|x| \le 1$), выраженный в градусах в интервале $-90 \le$ ASIN(x) ≤ 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 $-\pi/2 < \text{ATAN}(x) < \pi/2$.

ATAND(x) - возвращает арктангенс вещественного аргумента x, выраженный в градусах в интервале $-90 < \text{ATAN}(x) < 90$.

ATAN2(y, x) - возвращает арктангенс (y/x), выраженный в радианах в интервале $-\pi \le \text{ATAN2}(y, x) \le \pi$. 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 $-180 \le \text{ATAN2D}(y, x) \le 180$. 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:

Аргумент	*Результат (в радианах)*	*Результат (в градусах)*
$y > 0$	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 работы с массивами

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, FPS содержит дополнительные подпрограмму SORTQQ сортировки одномерного массива и функцию BSEARCHQQ бинарного поиска в отсортированном массиве. Для их вызова необходимо подключить 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 элеменов. 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	*Тип 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 эти значения 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 программу.

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 те 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 программу.

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
real(4) rv                              ! а встречается число ke
n = size(a); n2 = n                     ! Запомним размер массива a
do i = 1, n                             ! Заполним массив a случайным
 call random(rv)                        ! образом
 a(i) = int(rv * 1000.0)
enddo
ada = loc(a)                            ! Адрес массива a
call sortqq(ada, n, SRT$INTEGER4)           ! Сортировка массива a
if(n .ne. n2 ) stop 'Ошибка сортировки'
k = bsearchqq(loc(ke), ada, n, SRT$INTEGER4)
if (k == 0) then
 print *, 'Элемент ke = ', ke, ' не найден'
 stop
endif
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
enddo
print *, 'Всего найдено элементов: ', i - k   ! 21
end
```

6.10. Справочные функции для любых типов

ALLOCATED(*array*) - возвращает .TRUE., если память выделена под массив *array*, .FALSE. - если память не выделена. Параметр *array* должен иметь атрибут ALLOCATABLE. Результат будет неопределенным, если не определен статус массива *array*. Результат имеет стандартный логический тип. Например:

```
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 *target* 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*.

6.11. Числовые справочные и преобразующие 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. Числовые справочные и преобразующие функции позволяют получать данные о параметрах модели описания заданной разновидности типа и о конкретных характеристиках числа в задающей его модели.

Каждая разновидность целого типа моделируется множеством

$$i = s * \sum_{k=1}^{q} w_k * r^{k-1}$$

где s это ± 1; q - положительное число (возвращается функцией DIGITS); r - основание в модели (возвращается функцией RADIX; $r = 2$); w_k - целое число ($r \le w_k \le r$).

Каждая разновидность вещественного типа моделируется множеством

$x = 0$

и

$$x = s * b^e * \sum_{k=1}^{p} f_k * b^{-k}$$

где s это ± 1; p - целое число (возвращается функцией DIGITS, $p > 1$); b - основание в модели (возвращается функцией RADIX; $b = 2$); e - целое число из отрезка $e_{min} \le e \le e_{max}$ (e_{min} возвращается функцией MINEXPONENT, e_{max} возвращается функцией MAXEXPONENT); f_k - целое число, $0 \le f_k < b$, кроме f_1, которое вдобавок не равно нулю.

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, в которой содержится *x* (то есть *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*) - для вещественного *x* возвращает наименьшее положительное число с таким же типом и параметром типа, как у *x*, которое в сумме с единицей в модели, включающей в себя *x*, дает результат, больший единицы. Возвращаемый функцией результат равен b^{1-p}:

Тип параметра x	*EPSILON (x)*
REAL(8)	2.22044049250313E-016
REAL(4)	1.192093E-07

EPSILON позволяет задать наименьшее из возможных приращений в итерационных задачах, например поиск корня функции. Задание приращений, меньших, чем возвращаемый EPSILON результат, делает алгоритм неработоспособным.

HUGE(*x*) - для целого или вещественного *x* возвращает наибольшее значение в модели, включающей в себя *x*. Тип и параметр типа результата такие же, как у *x*. Значение равно

$r^q - 1$ - для целого *x* и $(1 - b^{-p})b^{e_{max}}$ - 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 максимальный показатель степени в модели, включающей в себя *x*, то есть 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 минимальный показатель степени в модели, включающей в себя *x*, то есть 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)	6
REAL(8)	15
COMPLEX(4)	6
COMPLEX(8)	15

RADIX(*x*) - для целого или вещественного *x* возвращает стандартное целое, равное основанию в модели, включающей в себя *x* (*r* или *b*).

RANGE(*x*) - для целого, вещественного или комплексного *x* возвращает эквивалентный десятичный степенной диапазон значений в модели, включающей в себя *x*, то есть

INT(LOG10(*huge*)) для целых и

INT(MIN(LOG10(*huge*), -LOG10(*tiny*))) для вещественных *x*,

где *huge* и *tiny* - наибольшее и наименьшее числа в соответствующих моделях. Результат функции - стандартного целого типа.

Тип параметра x	*RANGE (x)*
INTEGER(1)	2
INTEGER(2)	4
INTEGER(4)	9
REAL(4)	37
REAL(8)	307
COMPLEX(4)	37
COMPLEX(8)	307

TINY(*x*) - для вещественного *x* возвращает наименьшее положительное значение в модели, включающей в себя *x*, то есть $b^{e_{min}}$. 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 модельного представления вещественных чисел

Функции такого рода возвращают значение, связанное с компонентами модельного представления фактического значения аргумента.

EXPONENT(x) - возвращает степенную часть (то есть e) в модели 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 дробную часть в модели представления x, то есть $x * b^{-e}$. 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)       !  4007FFFFFFFFFFFF
```

RRSPACING(x) - возвращает вещественное значение с таким же параметром типа, как у x, равное обратной величине относительного расстояния между числами в модели, содержащей число x, в области, близкой к x, то есть $|x * b^{-e}| * b^{p}$.

```
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 $x * b^{i}$, где i - целое число, b - основание модели для x ($b = 2$).

```
print *, scale(5.2, 2)           !       20.800000
```

SET_EXPONENT(x, i) - возвращает вещественное значение с таким же параметром типа, как у x, равное $x * b^{i-e}$, где i - целое число, b - основание модели для x ($b = 2$), e = EXPONENT(x).

SPACING(x) - возвращает вещественное значение с таким же параметром типа, как у x, равное абсолютному расстоянию между числами в модели, содержащей число x, в области x, то есть b^{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
enddo
kr = 300
do while(selected_real_kind(r = kr) > 0)
 kr = kr + 1
enddo
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, $k = 0, 1, \ldots, s - 1$. 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 равны 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 целы-

ми 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 оба равны единице или оба равны 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 *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 *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 *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 $|shift| \le$ 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 *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 $0 < size \le$ BIT_SIZE(i); $|shift| \le size$ или $|shift| \le$ 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 *i* равен нулю, и, наоборот, бит результата равен нулю, если соответствующий бит *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. *From, frompos, len* и *topos* - целые параметры с видом связи IN; *to* - целый параметр с видом связи OUT. Значения параметров должны удовлетворять условиям: $len \ge 0$, $frompos + len \le$ BIT_SIZE(*from*), $frompos \ge 0$, $topos \ge 0$, $topos + len \le$ 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. 6.1). Рассмотрим 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 пикселов, для координат может быть выбран 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. 6.1), 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.

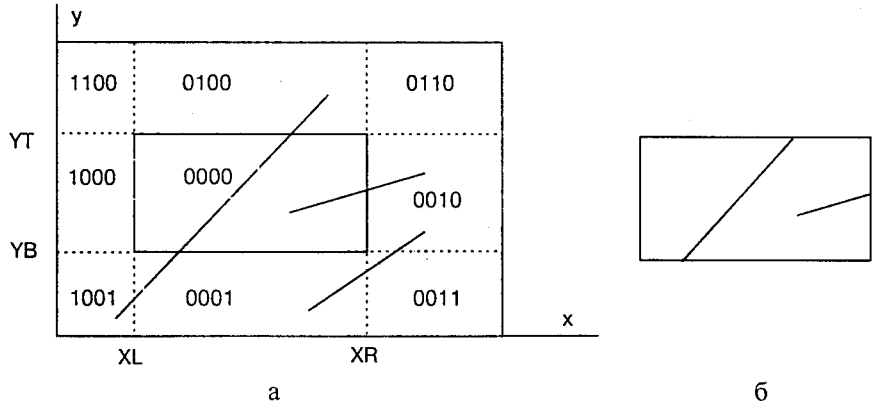

Рис. 6.1. Задача отсечения: а - коды областей; б - решение задачи отсечения

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.

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.

Запишем алгоритм отсечения в виде линейной схемы.

Интерфейс.

Входные данные.

XL, XR - x - координаты правой и левой границ окна вывода;

YB, YT - y - координаты нижней и верхней границ окна вывода;

$x1$, $y1$ и $x2$, $y2$ - координаты вершин отрезка.

Выходные данные.

$x1$, $y1$ и $x2$, $y2$ - 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.

$c1$, $c2$ - коды областей расположения первой и второй вершин отрезка;

$x1$, $y1$ и $x2$, $y2$ - 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(c1, c2) = 0, то отрезок внутри окна; вывести x1, y1, x2, y2, иначе отрезок вне окна.
7°. Останов.

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 вершинами отрезка
  call swap(x1, x2);  call swap(y1, y2);  call swap(c1, c2)
 endif
 if ( x1 < XL ) then                  ! Отсечение слева
  y1 = y1 + dfloat(y2 - y1) * dfloat(XL - x1)/float(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
 endif
 c1 = code(x1, y1, XL, XR, YB, YT)                ! Код вершины 2 не изменился
enddo
if (fl) then                       ! Обмен вершинами отрезка
 call swap(x1, x2)
 call swap(y1, y2)
 call swap(c1, c2)
endif
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 '
endif
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 выраженим, вызовом функции или константой, то создается временная переменная, содержащая результат выражения, вызов функции или константу, и функция LOC возвращает ад-

рес 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. 11.10.)

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 мене 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
```

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 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.

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 второй главе, ограничившись, правда, наиболее часто употребляемыми конструкциями. Теперь же мы дадим полное описание всех управляющих конструкций (кроме приведенной в разд. 4.7 конструкции WHERE). При этом часть управляющих операторов и конструкций будет выделена в разряд нерекомендуемых или устаревших, сохраненных в FPS с целью преемственности с более ранними версиями.

Как и ранее, необязательные элементы операторов заключаются в квадратные скобки.

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, ELSE IF, ELSE, 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. 4.

Пример. GOTO используется для организации повторных действий, то есть 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
endif
...
```

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
 endif
 print *, 'Ошибка. Повторите ввод'
enddo
...
```

7.2. Оператор и конструкции IF

Дополнительно по сравнению с более ранними версиями MS Fortran 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, то есть 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 FPS существует еще один условный оператор - арифметический IF. Этот оператор относится к устаревшим свойствам Фортрана. Его описание дано в прил. 4.

7.2.2. Конструкция IF THEN ENDIF

[*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 ENDIF

[*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'
endif
```

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, то есть вычисляется значение ЛВ2, и если оно истинно, то выполняется БОК2. Если оно ложно, то управление передается на следующий ELSE IF, и так далее. Если ложны все ЛВ, то выполняется следующий за завершающим ELSE БОКn. 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 это имя 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 ENDIF, например:

```
if(ЛВ1) then
  БО1
else
  if(ЛВ2) then
    БО2
  else
    БО3
  end if
end if
```

```
if(ЛВ1) then
  БО1
else if(ЛВ2) then
  БО2
else
  БО3
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
 endif val_3
enddo
. . .
```

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 так далее. Если значение *тест-выражения* не найдено ни в одном списке и присутствует оператор CASE DEFAULT, то выполняется БОКn, 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 БОКi 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, должны обязательно 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
enddo
. . .
```

7.4. DO-циклы. Операторы EXIT и CYCLE

Простейшая конструкция DO

[*имя*:] DO
    БОК
  END DO [*имя*]

задает бесконечный цикл. Поэтому такой цикл должен содержать по крайней мере один оператор 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
enddo 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;

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
enddo
```

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. 4.

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 альтернативного возврата из подпрограммы.

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
enddo
print *, sa
```

Замечание. С позиций структурного программирования приведенные вычисления лучше выполнить, применив объединение условий и отказавшись от применения оператора CYCLE:

```
sa = 0
i = 1
```

```
do while(a(i) /= 0 .and.  i <= 100)
  if(a(i) > 5) sa = sa + a(i)
  i = i + 1
enddo
```

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
  enddo loop3
 enddo loop2
enddo loop1
if(a(i, j, k) >= 0) 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 появляется внутри цикла, то соответствующий ему оператор END IF должен быть внутри того же цикла;
- если оператор SELECT CASE появляется внутри цикла, то соответствующий ему оператор END SELECT должен быть внутри того же цикла;
- если оператор WHERE появляется внутри цикла, то соответствующий ему оператор END WHERE должен быть внутри того же цикла.

Замечание. Последние три запрета легче контролировать, если соблюдать при записи программы правило "рельефа".

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 enddo
sa = 0
do 22 k = 1, 100
 if(a(k) .lt. 0) sa = sa + a(k)
22 enddo
! 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 enddo
! 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
! Вариант 6
k = 1
sa = 0
do 26 while (k .le. 100)
 if(a(k) .lt. 0) sa = sa + a(k)
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;
- оператор и конструкция WHERE;
- 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)
enddo
```

В последнем примере вместо встроенного DO-цикла $(a(i, j), j = 1, n)$ в списке вывода присутствует сечение массива, которое так же, как и встроенный DO-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
enddo
```

нельзя заменить присваиванием

```
a(2:n) = a(1:n - 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:

1) выводится *сообщение*;
2) 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 'Ошибка при открытии файла.'
endif
```

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 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.

```
pause 'Введите DIR для просмотра или нажмите Enter для возврата.'
read(*, '(a)') filename
open(1, file = filename)
```

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 головной 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 функцией значение. В программной единице BLOCK DATA переменным именованного *common*-блока можно было задать начальные значения.

Фортран 90, сохраняя все возможности Фортрана 77, дополнительно позволяет:

- создавать модули, содержащие глобальные данные и модульные процедуры;
- создавать внутренние процедуры, расположенные внутри головной программы, внешней или модульной процедуры.

Эти нововведения дополнительно позволяют:

- использовать различным программным единицам объявленные в модуле глобальные данные;
- использовать объявленные в модуле данные во всех процедурах этого модуля;
- использовать один и тот же объект данных во внутренней процедуре и в носителе этой процедуры.

Таким образом, в FPS можно создать такие программные единицы, как:

- головная программа;
- модули;
- внешние процедуры;
- внутренние процедуры;
- программные единицы 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*-блоки 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. 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 FPS можно задать *внешние, внутренние* и *модульные процедуры.*

Внешняя процедура является самостоятельной программной единицей и существует независимо от использующих ее программных единиц. В Фортране к любой внешней процедуре можно обратиться из головной программы и любой другой процедуры.

Модули и внешние процедуры могут компилироваться отдельно от использующих их программных единиц.

Внутренняя процедура задается в головной программе, внешней или модульной процедуре. Головная 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 в отличие от Фортрана 77 поддерживает рекурсивные вызовы процедур, то есть 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, SEED, RANDOM.

Подключаемые процедуры и *модули* находятся в ранее созданных библиотеках. Перечень и описание таких процедур дан в прил. 3.

Также с FPS поставляются математические библиотеки и библиотеки математической статистики.

Все поставляемые с FPS процедуры по умолчанию доступны для компилятора. Для их использования в программной единице следует подклю-

чить к ней модуль, содержащий используемые при работе с процедурами глобальные данные и интерфейсы. Такое подключение выполняется 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 ее имя в опции компилятора /link.

Процедуры и модули проекта создаются программистом для реализации конкретного проекта. На начальных стадиях разработки создаваемые процедуры и модули хранятся в *исходном коде*. Размещаются они, как правило, в разных файлах. Если в файле находится несколько процедур или модулей, то порядок их размещения в этом файле произвольный. Однако текст модуля должен быть размещен до первой ссылки на этот модуль. В общем случае в одном файле могут существовать внешние процедуры, модули и головная программа. Однако 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 попрежнему 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 программирования, например поиск экстремума функции;
- представляет логически самостоятельную задачу, например В/В данных и контроль ошибок В/В.

Также в виде процедур оформляется повторяющийся в программе более одного раза код. Реализация 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 решением проблемы 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 работнику 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 FPS

В FPS программа рассматривается как проект. Типы возможных в FPS проектов приведены в табл. 8.1.

Таблица 8.1. Типы проектов FPS

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 головной программы. Два последних - библиотечные проекты без головной программы.

Тип проекта задает вид генерируемого кода и некоторые опции проекта. Например, он определяет опции, которые использует компилятор, библиотеки, применяемые компоновщиком, задает по умолчанию размещение выходных файлов, константы проекта и так далее. Задание типа проекта выполняется при создании нового проекта.

Создание нового проекта в среде FPS выполняется в результате использования цепочки: File - New - Project Workspace - OK - выбрать тип проекта, например QuickWin Application, - ввести имя проекта - задать расположение проекта на диске - Create. После нажатия Create будет создана директория (папка), имя которой совпадает с именем проекта.

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 генерацию рабочего, не содержащего отладочного кода, EXE-файла. В любой из этих модификаций 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 Default Configuration... - выбор конфигурации - OK.

Чтобы закрыть проект, следует выполнить цепочку File - Close Workspace.

Существующий проект открывается в результате выполнения цепочки File - Open Workspace - выбрать файл проекта - Open.

В проект можно добавить как уже существующий, так и новый файл с исходным текстом или данными. Для добавления файла в открытый проект следует использовать последовательность Insert - File Into Project - выбрать файл или внести в поле "Имя файла" имя нового файла, например subt.f90 - Add.

Можно предварительно создать новый файл, а затем, используя Insert - File Into Project, добавить его в проект. Создание нового файла выполняется так: File - New - Text File - OK. Далее после ввода данных сохраним файл: File - Save - выбрать на диске директорию для записи файла - задать имя файла, например: subt.f90 - сохранить.

Для удаления файла из открытого проекта достаточно выбрать этот файл в окне File View и нажать Del или использовать цепочку Edit - Delete.

В результате выполнения цепочки Build - Build *имя файла* создается файл, имеющий имя проекта и расширение EXE.

При построении исполняемого (EXE) файла возможны ситуации:

- проект включает все используемые в нем файлы;
- часть файлов не включена в проект.

В первом случае в проекте указан полный путь к каждому файлу, который и используется компилятором.

И в первом и во втором случаях для включения модуля (независимо от того, включен файл с текстом модуля в проект или нет) используется оператор USE.

Во втором случае компилятор выполняют поиск всех не включенных в проект файлов.

Если в проекте нет файла с текстом модуля, то компилятор выполняет поиск файла с расширением MOD, который, в свою очередь, ссылается на файл с исходным текстом модуля. Имя файла с расширением MOD совпадает с именем подключаемого модуля.

Если файл не является модулем, а содержит процедуры или фрагмент программы, то его включение выполняется оператором INCLUDE или метакомандой $INCLUDE. Оператор (метакоманда) могеж содержать полное имя включаемого файла (то есть имя и путь к файлу). Если же полное имя файла не указано, то компилятор ищет файл в последовательности:

1. Поиск в директории, содержащей исходный файл.

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.

В такой же последовательности выполняется и поиск модуля.

При нахождении включаемого файла (модуля) поиск прекращается. В случае неудачи генерируется сообщение об ошибке.

В среде FPS директории для поиска включаемых файлов и модулей можно установить, выполнив цепочку Build - Settings - Fortran - Preprocessors - занести в поле INCLUDE and USE paths пути к файлам - OK. Путь к файлам завершается слэшем. При наличии нескольких путей они разделяются запятыми, например: myfiles/, mylib/. Заданные опции компилятора отображаются в поле Project Options.

8.4. Головная программа

Любая программа имеет одну головную 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.

Головная программа не может содержать операторы SUBROUTINE, FUNCTION, MODULE, 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 RETURN и ENTRY не могут появляться среди исполняемых операторов головной программы.

Выполнение программы всегда начинается с первого исполняемого оператора головной программы. Оператор END, если в результате вычислений на него передается управление, завершает выполнение программы. Оператор может иметь метку, используя которую можно перейти на END от других исполняемых операторов. Нормальное завершение программы может быть также выполнено оператором STOP, который может быть размещен как в головной программе, так и в процедуре.

8.5. Внешние процедуры

В Фортране могут быть определены два типа внешних процедур: *подпрограммы* и *функции*. *Функция* отличается от *подпрограммы* тем, что вызывается непосредственно 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. В FPS в предложении RESULT можно задать для результата другое имя.

Оператор CONTAINS в процедуре играет такую же роль, какую он играет и в головной 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 головной 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, DIMENSION, EXTERNAL, INTRINSIC, PARAMETER, POINTER, PRIVATE, PUBLIC, SAVE, TARGET. 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.

Модуль, используя оператор USE, можно включить:

- в головную программу;
- во внешнюю процедуру;
- в другой модуль.

При включении модуля в другой модуль следует следить, чтобы модуль не ссылался сам на себя ни прямо, ни косвенно через другие модули.

Пример.

```
module testmod
 integer, save :: a = 1, b = 1      ! a, b и c доступны в smod и smod2
 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
    end subroutine smod2
  end subroutine smod
end module testmod

program prom
 use testmod
 print *, 'prom_1:', a, b
 call smod(a)
 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:     0     1
smod2 :     2     2
smod :      3     3
prom_2:     3     3
osub :      4     4
prom_3:     5     5
```

Из примера видно, что изменения объявленных в модуле *testmod* переменных *a* и *b* передаются в программные единицы, использующие этот модуль. Такой способ обмена данными называется *use-ассоциированием*. В примере *use*-ассоциирование использовано для передачи данных в головную программу и внешнюю подпрограмму *osub*. Также видно, что значение переменной *b*, объявленной в модуле *testmod*, известно модульной подпрограмме *smod* и внутренней подпрограмме *smod2*. Такой способ передачи данных называется *ассоциированием через носитель*. Благодаря ассоциированию через носитель, модульная подпрограмма *testmod* имеет доступ к переменной *c*, а внутренняя подпрограмма *smod2*, носителем которой является модульная подпрограмма *smod*, имеет доступ к локальной переменной *c2* подпрограммы *smod*. Также в примере головная программа prom, модульная подпрограмма *smod*, внутренняя подпрограмма *smod2* обмениваются данными через параметр *d*. Такая передача данных называется *ассоциированием параметров*.

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 от-

компилированными 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.

Однако оператор USE позволяет:

- ограничить доступ к объектам модуля за счет применения параметра ONLY;
- использовать в программной единице для объектов модуля другие имена.

Например, приведенный выше модуль *testmod* можно подключить 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 задаваемый оператор, задаваемое присваивание, родовое имя или 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*.

Все используемые при переименовании локальные имена должны отличаться друг от друга и от локальных имен использующей модуль программной единицы. Локальное имя программной единицы должно обязательно появиться в операторе объявления типа. В противном случае оно будет неотличимо от выбираемого из модуля объекта. Чтобы избежать таких недоразумений, в программную единицу, содержащую оператор USE, следует вводить

IMPLICIT NONE

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, таким способом:

```
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*-группу. Однако нельзя задать атрибут PUBLIC объекту включаемого модуля, если в этом модуле объект имел атрибут PRIVATE.

Пример. 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 атрибута 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 атрибута 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 таков:

[RECURSIVE] SUBROUTINE *имя подпрограммы* &
[([*список формальных параметров*])]

Общий вид оператора заголовка функции следующий:

[*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 головной 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 фор-

мальных 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, C или VARYING), 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 к ней интерфейс и использовать с ней Microsoft-атрибуты DLLEXPORT или DLLIMPORT (прил. 2).

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 метакоманда !MS$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 метакоманда !MS$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 вызвана функция. Результирующая переменная после выполнения функции может быть не определена, если она является ссылкой и вызов функции выполнен не из выражения (разд. 3.11.7). Обычно результирующая переменная получает значение в результате присваивания. Но это необязательно. Например, в случае символьной функции результирующая переменная может быть определена в результате выполнения оператора

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. 3.11.7). 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 при-

вести 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                  ! элементом
   finel = c(i)
   return                            ! Возвратим последний отрицательный элемент
  endif
 enddo
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
  enddo
  nc = nc + 1
  c(nc) = va
 enddo 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 параметрами). Так, в нашем примере при первом вызове подпрограммы *fobc* фактическому параметру a соответствует формальный параметр *a*, фактическому параметру *b*1 - формальный параметр *b* и так далее. 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, буквальные и именованные константы, простые переменные, массивы и их сечения, элементы массивов, записи, элементы записей, строки, подстроки, процедуры и встроенные функции. В случае подпрограммы именем фактического параметра может быть и метка. Фактические параметры могут также иметь атрибуты POINTER или TARGET.

Формальными параметрами могут быть переменные (полные объекты), процедуры и звездочка (*).

Фактические и формальные параметры могут иметь атрибуты 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

Фактические параметры	*Формальные параметры*
Простая переменная	Простая переменная
Элемент записи	Переменная

Запись	Запись
Строка	Строка
Подстрока	"
Массив, сечение массива или элемент массива	Массив или простая переменная
Процедура	Процедура
Выражение	Переменная
Константа	"
*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 массив, то формальным параметром может быть массив заданной формы, или перенимающий форму массив, или перенимающий размер массив (разд. 4.9).

4. 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. То есть 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 выполняемых программой действий. На самом деле в FPS вид связи формального параметра можно задать явно, использовав для него 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(na), 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, константой или переменной;
- OUT - формальный параметр является выходным. При вызове процедуры 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 фор-

мальным 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 имеет возможности, например, проверить, соответствуют ли фактические и формальные параметры так, как это им положено.

Вызовы внешних процедур с неявным интерфейсом допустимы и в FPS. 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 в FPS 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. То есть 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, интерфейс к этой процедуре является явным.

Также все встроенные процедуры заведомо обладают явным интерфейсом.

В случае внешней процедуры в FPS 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
end subroutine sub1

program idem
interface                             ! Интерфейс к подпрограмме sub1
 subroutine sub1(int1, int2, real1, real2)
  integer :: int1, int2
  real, optional :: real1, real2
 end subroutine sub1
end interface
```

В FPS тело интерфейса может быть задано к внешним процедурам, написанным на других языках программирования, например на ассемблере или СИ (прил. 2).

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. 3.11.7);
- 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 используются при:

- вызове процедуры с ключевыми словами (разд. 8.11.4);
- использовании задаваемого присваивания (разд. 8.12.2);
- использовании задаваемых операций (разд. 8.12.2);
- использовании в вызове родового имени (разд. 8.12).

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
 endif
 if(present(me)) then
  mval = me
 else
  mval = size(array)
 endif
 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 к процедуре, 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 целое, а не подобъект.

В Фортране 77 расположение соответствующих формальных и фактических параметров в списке параметров должно совпадать. (То есть пер-

вый формальный параметр ассоциируется с первым фактическим и так далее.) В FPS 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(array = a, sig = 1, me = m)
result = npe(a, sig = 1, me = m)
result = npe(a, sig = 1)
```

В первом случае все параметры вызываются с ключевыми словами. Во втором и третьем - первые два параметра являются позиционными (то есть их положение совпадает с положением соответствующих формальных параметров), поэтому их вызов может быть выполнен без ключевых слов. В третьем случае не задан третий 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, компилятор FPS не выдаст ни одного сообщения или предупреждения об ошибке. Однако это совсем не означает, что программа отработает правильно. В общем случае результат непредсказуем.

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                   ! T         7.000000
end

logical function flo(z)
 z = 100
 flo = .true.
end function
```

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, совершенно очевидно, что следует избегать подобных вызовов функции. Действительно, если бы значение *x* было равно 3.0, то после вычисления выражения мы получили бы совсем иное значение для *z* - число 100.0.

Другой пример неполного вычисления выражения:

```
character(len = 2) stre, st1, st2
character(len = 2) stfun            ! stfun - символьная функция
...                                 ! Вызов stfun не будет выполнен,
stre = st1 // stfun(st2)            ! поскольку длина результата равна st1
```

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 в FPS 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 в FPS 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 (len = max(len(ch1), len(ch2))) chmax, ch1, ch2
  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
 endif
end function inmax

function remax(re1, re2)
 real (4) remax, re1, re2
```

```
 . . .
end function remax
function chmax(cha1, cha2)
 character (len = max(len(cha1), len(cha2))) chmax, cha1, cha2
 if(lge(cha1, cha2)) then
  chmax = cha1
 else
  chmax = cha2
 endif
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
```

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 в общем случае иметь разное число параметров. Также в FPS 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 подключаемый модуль, родовым именем. Тогда с помощью этого имени могут быть вызваны все охватываемые им процедуры.

Все процедуры, объединяемые под одним родовым именем (описанием), 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 условия 2:

```
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
```

8.12.2. Перегрузка операций и присваивания

В FPS можно расширить область применения встроенной операции. 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
  type (ire) funir                   ! операцию функции должны
  type (ire), intent(in) :: rec1, rec2 ! иметь вид связи in
  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 веденной операции может быть таким:

```
t3 = t1 .plus. t2 .plus. t3
```

Как и встроенная, вновь вводимая операция может быть распространена на разные типы операндов.

В FPS 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 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-
 type(icha) :: t                   ! ассоциирования 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 тоже 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

INTERFACE OPERATOR (*определяемая операция*)

INTERFACE 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 процедуры*]]

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 головной программы, встроенных и внешних процедур, модулей, *common*-блоков, BLOCK DATA. Однако локальный объ-

ект 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 подключаемого 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 имен подключаемых модулей и локальных имен самой программной единицы. Однако доступные через *use*-ассоциирование имена модулей 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 пу-

тем *use*-ассоциирования, или являющегося локальным формальным параметром или результирующей переменной.

Пример.

```
real :: x = 1.0, y = 1.0, z = 1.0
 call decar(x)
 print '(3f5.2)', x, y, z          ! 5.0  1.0  5.0
 contains
  subroutine decar(x)
   real x, y                      ! Локальная переменная у подпрограммы decar
   x = 5; y = 5; z = 5            ! закрывает локальную переменную у 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
```

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. В FPS имена объектов разделяются на *глобальные*, *локальные* и *операторные*.

Глобальными в FPS являются имена головной 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 функция также становится недоступной в этом блоке видимости, но введенное имя трактуется как глобальное имя внешней функции.

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. Такой же эффект вызовет и использование внутренней функции с именем, например, *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 во встроенных DO-циклах операторов DATA и конструкторах массивов. Областью видимости формальных параметров операторной функции является оператор задания этой функции. Областью видимости переменной встроенного DO-цикла, которая должна быть целого типа, является этот цикл. Параметры встроенного DO-цикла операторов B/B не являются операторными, а относятся к локальным и могут быть вещественного типа. Например:

```
real c(100)
z(x, y) = sin(x) * exp(-y)
b = 55.0
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                     !       1.00000
print *, k                     !       55
```

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 слэшами. Например:

```
common /vab/ a, b
real vab                       ! Имена переменной 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. Головная программа и каждая процедура имеют свой независимый набор меток. Оператор END носителя может иметь метку. Если в таком носителе есть внутренние проце-

дуры, 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. В программе, применив оператор COMMON, можно создать общую область памяти, на которую можно ссылаться из 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 три 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* головной 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)
 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 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. 4).

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 (то есть 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 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 масси-

ве. 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 слэши 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, заданным для этих элементов в операторе COMMON.

Длина общей области равна числу байт памяти, необходимых для размещения всех ее элементов, включая расширения за счет EQUIVALENCE-ассоциирования (прил. 4). 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.

FPS 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.

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 Microsoft-атрибута: ALIAS и C. Не следует из-за приведенных проблем

выравнивания использовать *common*-блок для доступа к структурам СИ, применяя вместо определение типа с Microsoft-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. Более 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 слэшами, 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 должн 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 FPS все (кроме автоматических) объекты по умолчанию имеют атрибут SAVE. Поэтому явное задание этого атрибута именованному *common*-блоку полезно при создании переносимых на разные 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 /vab/ 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
 x = 5.0; y = -5.0
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

FPS 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
 endif                             ! выполняется до тех пор, пока не
end                                ! выполнены все замены sub1 на sub2
```

Если функция содержит прямую рекурсии, то есть 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, а имя результата fav
 if(n <= 1) then
  fav = 1
 else
  fav = n * ifact(n - 1)              ! Рекурсия продолжается пока n > 1
 endif
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 так далее. 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 использование 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 еди-

нице, 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 используется 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/(1.2*arctgx + Цx + 1) (ответ: x = 0.5435)

и

x = (e-x + Цex + 3.7) / 3 (ответ:x = 0.8614)

Алгоритм метода

1. Задать начальное приближение x0, например, положить x0 = (a + b)/2.
2. Положить x = f(x0).
3. Если |x - x0| > eps, то выполнить п. 4, иначе выполнить п. 5.
4. x0 = x; x = f(x0); перейти к п. 3.
5. Принять в качестве решения последнее значение переменной x.

Проиллюстрируем метод простых итераций на рис. 8.1.

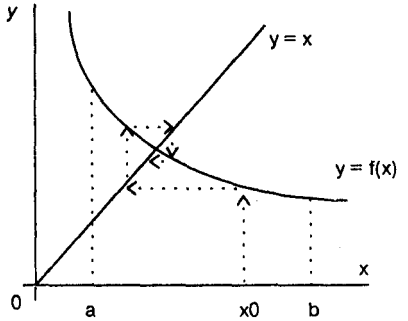

Рис. 8.1. Метод простых итераций

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
real function fx2( x )
 real x
 fx2 = (exp(-x) - sqrt(exp(x)) + 3.7) / 3.0
end
! 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
 x0 = (a + b)/2.0
 x = fx(x0)
 do while(abs(x - x0) .gt. eps)
  x0 = x
  x = fx(x0)
 enddo
 root = x
end
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 в
!  end function fx1                     ! 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
```

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*

список 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 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 "Тип функции" указан тип, который 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*	*Специфическое имя*	*Тип аргумента*	*Тип функции*
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)

Описание функции	*Форма вызова с родовым именем*	*Специфическое имя*	*Тип аргумента*	*Тип функции*
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 градусах)	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 градусах)	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 градусах)	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 градусах)	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 градусах)	ACOSD(X)	ACOSD DACOSD	Real Real(8)	Real(4) Real(8)

Описание функции	*Форма вызова с родовым именем*	*Специфическое имя*	*Тип аргумента*	*Тип функции*
Арктангенс	ATAN(X)	ATAN DATAN	Real Real(8)	Real(4) Real(8)
Арктангенс (результат в градусах)	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 градусах)	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	Character	Integer(4)

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*	*Специфическое имя*	*Тип аргумента*	*Тип функции*
Преобразование к целому типу	INT(A)	INT IFIX IDINT	Real, Cmp Real(4) Real(8)	Integer Integer(4) Integer
Преобразование к вещественному типу	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(a1, a2, ...)	MAX(A1, A2, ...)	MAX0 AMAX1 DMAX1 AMAX0 MAX1	Integer Real Real(8) Integer Real	Integer Real Real(8) Real Integer
min(a1, a2, ...)	MIN(A1, A2, ...)	MIN0 AMIN1 DMIN1 AMIN0 MIN1	Integer Real Real(8) Integer Real	Integer Real Real(8) Real Integer

Пример. Построить графики функций *sinx* и *cosx* на отрезке $[-\pi, \pi]$. Для работы в графическом режиме необходимо создать проект как приложение QuickWin или Standard Graphics. Для доступа к процедурам графической библиотеки подключается 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 GETWINDOWCONFIG. Назначение использованных графических процедур можно понять из размещенного в тексте программы комментария. Их подробное описание приведено в гл. 12.

```
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, YE/4, 3*XE/4, 3*YE/4)
! 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)          ! Оси координат - белым цветом
   call moveto(int2(XE/4 - 10), int2(YE/2), xy)
   status2 = lineto(3*XE/4 + 10, YE/2)      ! Ось x
   call moveto(int2(XE/2), int2(YE/4 - 10), xy)
   status2 = lineto(XE/2, 3*YE/4 + 10)      ! 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
```

```
  enddo
 end subroutine curve
end
```

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
endif
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
 enddo
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:

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 атрибута POINTER. В случае текстового результата результирующие переменные должны быть одной длины.

При работе с оператором ENTRY следует соблюдать 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 ENDIF), 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
endif
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 FPS по умолчанию переменные, как правило, являются *статическими*. То есть 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 FPS в процедуре и модуле можно сделать переменную автоматической, присвоив ей атрибут AUTOMATIC. Атрибут AUTOMATIC является расширением FPS над стандартом Фортран 90 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. Следовательно, 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.

```
call atav(2, 3)
call atav(4, 5)
end

subroutine atav (m, n)
 automatic                    ! Переменные объявляются автоматическими
 integer :: m, n, a, b = 2    ! неявно
 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 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 слэшами. 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 головной программе не имеет никакого действия. Если *common*-блок задан в головной 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 эквивалентен ранее приведенному атрибуту SAVE и атрибуту STATIC языка СИ. Значения статических переменных сохраняются после выполнения оператора RETURN или END. Напомним, что в FPS по умолчанию переменные (кроме динамических) размещены в статической памяти. Для изменения правил умолчания используются атрибуты ALLOCATABLE, AUTOMATIC и POINTER. Синтаксис оператора STATIC:

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 слэшами (/).

Пример.

```
integer :: ng = -1
do while ( ng )                          ! Цикл завершается при ng = 0
 call sub1(ng, ng + ng)
 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 .eq. -1) then
    n2 = iold
    n3 = iold
   endif
   print *, 'new: ', inew, '  n2: ', n2, '  n3: ', n3
  end subroutine
end
```

8.24. 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 $z = \text{siny} * e^{-x}$.

```
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
enddo
x = x + dx
 write(*, *)                           ! Переход на новую строку
enddo
end
```

Замечание. Для вывода без продвижения на новую строку используется преобразование обратного слэша (\).

8.25. Оператор 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 метакоманды $INCLUDE или оператора INCLUDE, имеющего 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.

Оператор INCLUDE вставляет содержимое текстового файла в то место программной единицы, где он расположен. (Оператор 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 оператора INCLUDE.

Включаемый файл может содержать другие операторы 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 операторов INCLUDE.

Первая строка включаемого файла не должна быть строкой продолжения, а его последняя строка не должна содержать переноса. Перед оператором не может быть поставлена метка.

В FPS *include*-файлы рассматриваются как избыточное средство языка и могут быть практически полностью и с большим эффектом заменены модулями. Модули обеспечивают доступ не только 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.

8.26. Порядок операторов и метакоманд

Операторы и метакоманды в программных единицах FPS должны появляться в приведенном в табл. 8.5 порядке.

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 8.5. Последовательность операторов и метакоманд FPS

<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="12">$[NO]DEBUG
$[NO]DECLARE
$DEFINE, $UNDEFINE
$IF, $IF DEFINED
$ELSE, $ELSEIF, $ENDIF
$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>PARAMETER</td><td rowspan="10"></td></tr>
<tr><td>IMPLICIT</td><td rowspan="3"></td></tr>
<tr><td>Определения производных типов</td></tr>
<tr><td>Интерфейсные блоки</td></tr>
<tr><td>Операторы объявления типа</td><td>PARAMETER</td><td>ENTRY</td></tr>
<tr><td>Операторы объявления</td><td>DATA</td><td>FORMAT</td></tr>
<tr><td>Операторные функции</td><td colspan="2" rowspan="2"></td></tr>
<tr><td>Исполняемые операторы</td><td>DATA</td></tr>
<tr><td colspan="3">CONTAINS
Внутренние и модульные процедуры
END</td></tr>
</table>

Таблица 8.6. Операторы программных компонентов

Операторы	*Головная 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 [2]. 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 В/В 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 будут 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 содержит 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. Необходимость его присутствия вытекает из следующего: по умолчанию в Фортране при форматном выводе первая позиция строки вывода предназначена для простановки символа управления кареткой печатающего устройства. Возмлжные 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 также подсоединить внешнее устройство, задав опцию 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)
```

Замечание. В среде FPS 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 хвостовых 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*.

```
character(78) :: title = 'Пример заголовка'
character(20) form                    ! Строка формата вывода
integer(1) tl, n, sl /80/
tl = len_trim(title)                  ! Длина заголовка без хвостовых 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
```

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.

Замечание. 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).

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) form(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 дескрипторе 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
enddo                              !     1000
end                                !    10000
```

Целочисленное выражение может быть любым допускаемым в FPS 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 формата 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):

```
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 поля с данными, которые передаются из списка вывода. Каждый элемент списка вывода создает в файле поле, размер которого определяется форматом вывода.

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. Раз-

мер 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 /(1x, 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. В противном случае возникнет ошибка ввода. При вводе с клавиатуры для завершения оператора READ придется ввести 31 значение, придерживаясь установленного спецификацией *fmt* формата.

Преобразование слэша (/) в спецификации формата обеспечивает переход файлового указателя на начало новой строки.

Всего в файле a.txt должно быть не менее 18 записей, например таких (символ □ использован для обозначения пробела):

```
□111 222 333 444
□11 12 13 14 15 16 17 18 19 20
□1111.11 2222.22
□строка1
...
□строка 15
```

На экран будет выведена точная копия файла a.txt.

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 атрибута 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 циклический список оператора DATA. Циклический список также называют *встроенным DO-циклом.*

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. + cosx/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, для определения 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+cosx/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
  enddo                               ! Вывод строки таблицы
  write(*, '(2x, f5.2, 9(1x, f7.2))') x, z(:k)
  x = x + dx
 enddo
 write(*, 1) ('_', k = 1, 80)
1 format(80a1)                        ! Формат вывода горизонтальной линии
end program
```

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);
- дескрипторы управления;
- 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.n*. 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 ДП не содержат дескриптор \ или $ или в операторе В/В не задана опция ADVANCE = 'NO', обеспечивающие 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 вывода 11 элементов. Переменная *k* выберет ДП I8, *n* - I5, *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 применяются, 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*]	"	Вещественное число в E-форме
EN*w.d*[E*e*]	"	" " " "
D*w.d*	"	Вещественное число 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 ($d < w$).

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, то есть расположен левее следующего по значимости байта).

Дескриптор Z может быть применен с символьными данными, если длина строки не превышает 130 символам. 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 $w \le 8*n$ (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 $w \ge 8*n$ (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 хвостовые пробелы трактуются как нули, если в операторе OPEN задана опция 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 управляемым списком-вводом, 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, то есть 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*	$\|p\| \le (10e) - 1$	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, то есть 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, то есть 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 А 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 так далее.

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 хвостовыми пробелами.

Выводимые с применением дескриптора А данные выравниваются по правой границе поля, хвостовые 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. То есть 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*Ee 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 P и сразу следующими за ним дескрипторами F, E, EN, ES, D или G, например: 1X, 2P F9.6;
- перед или после дескрипторов апострофа ('), кавычки ("), обратного слэша (\) или двоеточия (:), например: 1x, I3, ' ' B8 \;
- перед и после слэша, например: 1x, 2I5, 2(/ 2F5.2).

Таблица 9.7. Неповторяющиеся дескрипторы преобразований

Форма	*Имя*	*Назначение*	*Использование*
Строка	Преобразование апострофа	Передает строку текста на внешнее устройство	Вывод
nH	Преобразование Холлерита	Передает n символов на внешнее устройство	Вывод
Q	Преобразование опроса	Возвращает число непрочитанных символов записи	Ввод
Tn, TLn, TRn	Преобразование позиции	Спецификация позиции в записи	В/В
nX	Преобразование позиции	Спецификация позиции в записи	В/В
SP, SS, S	Преобразование знака плюс	Управление выводом знака плюс	Вывод
/	Преобразование слэша	Переход к следующей записи и простановка символов конца записи	В/В
\	Преобразование обратного слэша	Продолжение текущей записи (для тех же целей можно использовать знак $)	Вывод
:	Прерывание выполнения действия ДП	При исчерпании списка вывода прерывает выполнение ДП	Вывод
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 каж-

дый 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. Очевидно, что проще задать преобразование апострофа, поскольку при этом не требуется подсчитывать число передаваемых символов.

Преобразование опроса. Дескриптор Q получает 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, то есть применить ввод без продвижения, например:

```
integer k, nq                    ! Опция advance = 'no' задает
character(1) chr(80)             ! ввод 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 B/B 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 B/B 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 чисела. 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 слэша. В текущей записи *слэш* (/) указывает на конец подлежащих передаче данных.

При *вводе* слэш позиционирует файл за текущей записью.

При *выводе* слэш обеспечивает простановку символов конца записи и позиционирует файл за этими символами. Перед слэшем 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 слэша (/).

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 слэша. 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 слэш (\), 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 опция оператора B/B ADVANCE = 'NO', которая, 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, поля в случае BZ поля 1□□.2□3 и 100.203 эквивалентны.

Замечание. Пробелы, следующие после ДД 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 задана опция 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 в принятыми в Фортране соглашениями.

Управляемый списком В/В применяется при работе с текстовыми последовательными устройствами и не может быть использован при работе с неформатными файлами и файлами прямого доступа.

Различают два вида управляемого списком В/В: 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 В/В под управлением именованного списка:

WRITE (*u*, [NML =] *имя списка* В/В)

READ (*u*, [NML =] *имя списка* В/В)

u - номер устройства В/В.
имя списка В/В задается оператором NAMELIST.

Параметр NML= может быть опущен. Его присутствие обязательно, если заданы другие параметры оператора WRITE или READ, например END=.

Управляемый списком В/В 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 текстовый файл (или последовательное устройство, например экран) надо открыть с опциями 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 опция DELIM = 'QUOTE'. Отсутствие в операторе OPEN опции DELIM= эквивалентно заданию в нем опции 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 *$имя_списка*. Перечень принадлежащих именованному списку данных завершается слэшем (/) или знаком доллара ($) или амперсандом (&). 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
```

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 слэш, 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 - номер устройства В/В;

* - 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 слэша (/) 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, то есть 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, заключенные 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, слэши. Также невозможно разместить такую константу на нескольких строках.

- если длина символьной константы меньше или равна длины 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 слэш (/).

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);
- *логические данные* выводятся: T для .TRUE. и F для .FALSE.;
- *целочисленные данные* выводятся в соответствии с дескриптором I11;
- *вещественные* и *комплексные данные* в зависимости от значения выводятся в соответствии с дескрипторами F или E:
- если выводимое значение $1 \le val < 107$, то для одинарной точности используется преобразование F15.6, а для двойной - E24.15;
- если выводимое значение $val < 1$ или $val \ge 107$, то для одинарной точности используется преобразование E15.6E2, а для двойной - E24.15E3;
- *символьные данные* по умолчанию выводятся без ограничителей, однако после задания в операторе OPEN опций 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.

Пример. Вывод "длинной" константы.

```
character :: sub*10 = '1234567890', st*150 = ' '
do 1, i = 1, 15
1  st = trim(st) // sub
write(*, *) st
```

Замечание. Для управления выводом можно использовать СИ-символы: '\n'c - новая строка, '\r'c - возврат каретки, '\t'c - табуляция и другие:

```
character(4) year(5) /'1996', '1997', '1998', '1999', '2000'/
write(*, *) 3.55, '\t'c, 'pels', '\n\r'c, (year(i),' ', i = 1,5)
```

Результат:

```
3.550000 pels
 1996 1997 1998 1999 2000
```

10. Файлы Фортрана

10.1. Виды файлов. Файловый указатель

В Фортране различают два вида файлов: внешние и внутренние.

Внешний файл - поименованная область во внешней памяти ЭВМ.

Внутренние файлы: символьная строка (подстрока) или массив.

Внутренние файлы являются открытыми по умолчанию. Внешние файлы должны быть открыты (подсоединены к устройству В/В) оператором OPEN.

Файлы Фортрана подразделяются на файлы *последовательного* и *прямого* (произвольного) доступа. Внутренние файлы являются файлами последовательного доступа. Внешние файлы могут быть открыты как для последовательного, так и для прямого доступа.

Внешние файлы могут быть:

- *форматными* (текстовыми, ASCII);
- *двоичными* (бинарными);
- *неформатными*.

Двоичный и неформатный файлы содержат неформатные записи, то есть записи, создаваемые без преобразования данных. Файл не может одновременно содержать форматные и бесформатные 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 устройства либо после его закрытия, либо при нормальном завершении программы. При разделенном доступе внешний файл можно *заблокировать* (сделать недоступным для использования другим процессом).

Механизм доступа к файлу удобно иллюстрировать пользуясь понятием *файлового указателя* (ФУ). В результате выполнения операции над внешним файлом ФУ может находиться:

- до первой записи файла;
- между соседними записями файла;
- в пределах одной записи;
- после последней записи до специальной записи “конец файла”;
- на записи “конец файла”;
- после специальной записи “конец файла”.

ФУ указывает на “конец файла”:

- при отсутствии записей в файле;
- после чтения последней записи файла;
- после добавления новой записи в файл.

Факт перемещения ФУ на конец файла устанавливается функцией EOF, которая возвращает .TRUE., если ФУ позиционирован в конце файла или после конца файла, и .FALSE. - в противном случае.

10.2. Номер устройства

Для передачи данных Фортран использует устройства В/В. Такие устройства имеют номера. К устройствам В/В могут быть подсоединены внешние и внутренние файлы, а также физические устройства, например клавиатура, экран, принтер, параллельный порт. Устройства с номерами *, 0, 5 и 6 всегда существуют в каждой Фортран-программе. 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. Номером устройства является целочисленное выражение, значение которого должно находиться в интервале от 0 до 32767. После подсоединения и устройство и файл считаются открытыми. Доступ к файлу, после того как он открыт, выполняется по номеру устройства, к которому он подсоединен: все программные компоненты, ссылающиеся на один и тот же номер устройства, 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          ! Номер устройства - целочисленное 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'
```

В случае *внутреннего файла* номером устройства является имя строки, подстроки, символьного массива или его элемента.

Устройство не может быть одновременно подсоединено более чем к одному файлу, также и файл не может быть одновременно подсоединен более чем к одному устройству.

10.3. Внутренние файлы

Различают два основных типа внутренних файлов:

- символьную 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.

При работе с внутренними файлами можно использовать форматный В/В и В/В под управлением неименованного списка. Оператор PRINT, поскольку в нем отсутствует номер устройства, для работы с внутренним файлом неприменим.

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. Спецификатором внутреннего файла (номером устройства, к которому внутренний файл подсоединен) является имя соответствующей строки, подстроки или символьного массива. Часто внутренние файлы применяются для создания строк, содержащих смесь символьных и числовых данных (см. разд. 3.8.7), а также для простого преобразования число - строка символов и строка - число, например:

```
real :: a = 234.55
integer kb
character(20) st
write(st, *) a                    ! Преобразование число - строка
print *, st                       !    234.550000
read(st, '(i8)') kb               ! Преобразование строка - число
print *, kb                       !         23
end
```

10.4. Внешние файлы

В Фортране можно создать внешние файлы как последовательного, так и прямого доступа. Общие свойства таких файлов:

- для обеспечения доступа файл должен быть открыт (подсоединен к устройству В/В);
- при открытии файла по умолчанию ФУ позиционируется на первую запись файла или на конец файла, если в файле нет ни одной записи;
- каждая запись файла имеет номер; номер первой записи файла равняется единице;
- с любой позиции в результате выполнения оператора REWIND файл позиционируется на первую запись;
- с любой записи в результате выполнения оператора BACKSPACE файл позиционируется на одну запись назад (кроме рассмотренных в следующей главе специальных случаев);
- последней записью файла является специальная запись "конец файла".

В *последовательном* файле возможно лишь чтение и добавление записей. Модифицировать существующую запись непосредственно в последовательном файле нельзя. Для изменения записи возможен такой путь: прочитать все записи файла в массив; изменить в нем нужную запись; перейти на начало файла и записать массив в файл.

В файле *прямого доступа* любая запись может быть изменена. В этом и состоит основное отличие между файлами последовательного и прямого доступа. Кроме того, при включении в операторы В/В опции REC= файл прямого доступа может быть позиционирован вслед за указанной в этой опции записью. Непосредственное удаление записи из файла прямого доступа невозможно. Однако можно заменить ненужную запись на новую.

10.4.1. Двоичные файлы последовательного доступа

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. Число записей в двоичном файле равно числу переданных байт. Поэтому выполнение оператора BACKSPACE приведет к перемещению на 1 байт назад. Между записями последовательного двоичного файла FPS не проставляет символов (в отличие от неформатного, текстового или двоичного прямого доступа файла).

Ввод из двоичного файла значения переменной *val*, занимающей в оперативной памяти *n* байт, вызовет перемещение ФУ в двоичном файле на *n* записей (байт).

Если ФУ перед выполнением оператора вывода в последовательный файл находился на записи r_i, то запись r_i и все последующие записи в результате вывода будут "затерты" (заменены на выводимые записи). Это правило распространяется на все виды последовательных файлов.

Оператор OPEN, подсоединяющий файл к устройству для двоичного последовательного доступа, обязательно включает опцию FORM = *form*, где *form* - символьное выражение, вычисляемое со значением 'BINARY'.

Пример управления ФУ в двоичном последовательном файле:

```
integer(2) :: ia, ib, d(5) = (/ 1, 2, 3, 4, 5 /), i
real(4) :: a
character(3) ca                    ! Открываем двоичный файл
open(1, file = 'a.dat', form = 'binary')
write(1) 1.1, 2.2                  ! Запись в файл 8 байт
write(1) d                         ! Добавление в файл 10 байт
write(1) 'aaa', 'bbb', 'ccc'       ! Добавим в файл еще 9 байт
rewind 1                           ! Читаем 24 байта одним оператором read
read(1) a, a, (ia, i = 1, 5), ca, ca
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                          ! ФУ 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
```

Замечание. Любой внешний файл (неформатный, текстовый прямого и последовательного доступа) может быть открыт как двоичный, например с целью копирования данных.

10.4.2. Неформатные файлы последовательного доступа

В неформатные файлы, так же как и в двоичные, данные передаются без преобразований. Однако в неформатном файле в отличие от двоич-

ного 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, находясь на записи r_i, нельзя считать число байт, превышающее число байт в этой записи. Попытка такого чтения 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.

Так же как и в двоичных файлах, добавление новой записи при позиционировании ФУ на запись r_i приведет к удалению этой и всех последующих записей (к их замене на добавляемую).

Оператор OPEN, подсоединяющий файл к устройству для неформатного последовательного доступа, обязательно включает опцию FORM = = *form*, где *form* - символьное выражение, вычисляемое со значением 'UNFORMATTED'.

Пример управления ФУ в последовательном неформатном файле.

```
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 первой записи
write(1) d                         ! Вторая запись
write(1) 'aaa', 'bbb', 'ccc'       ! Третья запись
rewind 1
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 третьей записи
backspace 1                        ! Переход в начало второй записи
read(1) ib, ib
print *, a, ia, ca, ib             !     2.200000    4bbb    2
rewind 1
read(1) a                          ! Переход в начало второй записи
write(1) 'ghi'                     ! Замена второй и третьей 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
```

10.4.3. Текстовые файлы последовательного доступа

Текстовый 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. 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 последовательного текстового файла могут быть разной длины.

Если при форматном вводе не использованы дескрипторы \ или $, то после завершения ввода ФУ всегда перемещается на начало следующей записи (или позиционируется в конец файла), даже если были прочитаны не все поля текущей записи. Преобразования \ и $, а также применяемая в операторе READ опция ADVANCE = 'NO' обеспечивают ввод без продвижения. При форматном вводе число читаемых одним оператором ввода из текущей записи данных регулируется форматом ввода. В отличие от неформатного файла одним оператором ввода в принципе может быть прочитано произвольное число записей последовательного текстового файла.

В простейшем случае, открывая файл для последовательного форматного доступа, можно указать в операторе OPEN только номер устройства и опцию FILE = *имя файла*.

Пример управления ФУ в последовательном текстовом файле. По умолчанию последовательные файлы открываются с опцией 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 текстовый файл a.txt
open(1, file = 'a.txt')
write(1, 1) 1.1, 2.2
write(1, 2) d
write(1, 3) 'a', 'bc', 'def'
! После выполнения трех операторов вывода в файле будет 3 записи:
!   1.10   2.20
!    1    2    3    4
!   a  bc def
rewind 1                          ! Переход на первую запись
read(1, 1) a, b
read(1, 2) ia, ia, ia             ! Читаем из второй записи
! или  вместо двух последних операторов: read(1, *) a, b, ia, ia, ia
read(1, 3) ca, ca                 ! Читаем из третьей записи
backspace 1                       ! Переход на начало третьей записи
backspace 1                       ! Переход на начало второй записи
read(1, 2) ib, ib                 ! Пишем в ib второй элемент второй записи
write(*, *) a, ia, ca, ib         !     1.100000   3   bc   2
rewind 1                          ! Переход на первую запись
read(1, *)                        ! Переход на вторую запись
write(1, 3) 'ghij'                ! Все записи, начиная со второй, заменены на ghij
! После выполнения трех последних операторов имеем файл:
!   1.10   2.20
! ghij
end
```

10.5. Файлы прямого доступа

В Фортране можно создать двоичные, неформатные и форматные (текстовые) файлы прямого доступа. В файле прямого доступа все записи имеют одинаковую длину, задаваемую при открытии файла параметром RECL=. При этом в случае неформатного и текстового файлов число считываемых из файла байт и добавляемых в файл байт одним оператором В/В не должно превышать длины записи. В случае вывода недостающие байты записи будут содержать *null*-символы (CHAR(0)).

В файле прямого доступа можно перейти на любую запись r_i, выполнив оператор READ, в котором задан параметр REC = r_{i-1}. Список ввода такого оператора может быть пуст.

Переход на запись r_i в неформатном или двоичном файле:

```
read(2, rec = ri-1)
```

Переход на запись r_i в форматном файле:

```
read(2, '(a)', rec = ri-1)          ! Дескриптор формата - любой
```

В файле прямого доступа при выполнении оператора WRITE будет обновлена та запись, номер которой задан в параметре REC= оператора WRITE. Если параметр REC= в операторе WRITE отсутствует, то обновляется текущая запись. Все последующие записи файла будут сохранены. Если же ФУ указывает на конец файла, то выполнение оператора WRITE приведет к добавлению новой записи.

Чтобы удалить ненужные хвостовые записи файла прямого доступа, следует переместиться на первую удаляемую запись файла (это обычно выполняется оператором READ), а затем применить оператор ENDFILE.

Оператор OPEN, подсоединяющий файл *fname* для неформатного прямого доступа к устройству *u*, должен иметь опции

OPEN (*u*, FILE = *fname*, ACCESS = 'direct', RECL = *recl*)

где *recl* - целочисленное выражение, возвращающее длину записи файла.

Опция FORM = 'UNFORMATTED' в случае файла прямого доступа задается по умолчанию и может быть опущена. В случае форматного и двоичного файла прямого доступа опция FORM= является обязательной:

OPEN (*u*, FILE = *fname*, ACCESS = 'direct', FORM = 'formatted', RECL = *recl*)
OPEN (*u*, FILE = *fname*, ACCESS = 'direct', FORM = 'binary', RECL = *recl*)

Пример. В файле прямого доступа a.dat, содержащем 30 записей, удалить записи, имеющие номер, больший 15.

```
character(30) :: fn = 'a.dat'
integer :: iche, r = 15
open(1, file = fn, access = 'direct', form = 'formatted', &
       recl = 35, status = 'old', iostat = iche)
if (iche .ne. 0) stop 'Не могу открыть файл a.dat'
read(1, '(a)', rec=r, iostat=iche)   ! Переход на запись 16
if (iche .eq. 0) then                ! Если удалось прочитать запись r,
 endfile 1                           ! то проставляем метку конца файла
else
 write(*, *) 'Не могу прочитать запись ', r
endif
end
```

Замечания:

1. Для прогона файла к записи *r* можно использовать оператор READ без списка ввода.

2. При работе с текстовыми файлами прямого доступа возможен только форматный В/В. В/В под управлением списка недопустим. Также невозможен и В/В без продвижения.

3. Работая с файлами, следует контролировать коды завершения выполнения операторов OPEN, READ, WRITE и CLOSE.

Двоичный и неформатный файлы прямого доступа имеют структуру, совпадающую со структурой двоичного последовательного файла. Однако в двоичном файле прямого доступа длина записи определяется параметром *recl*, а не равна 1 байту, как в последовательном двоичном файле. После каждой записи двоичного и неформатного файлов прямого доступа проставляются разделители - два *null*-символа. Файлы прямого доступа могут быть открыты как двоичные последовательные файлы.

Отличия между неформатным и двоичным файлами прямого доступа:

- в двоичный файл прямого доступа можно записывать любое число байт, не обращая внимания на значение параметра *recl* (при этом, правда, длина записи все же определяется параметром *recl*);
- из двоичного файла одним оператором ввода можно считать больше байт, чем задано параметром *recl*.

Один и тот же двоичный или неформатный файл может быть открыт с разными значениями параметра *recl*.

Пример.

```
character(14) ch
open(2 ,file = 'a.bin', access = 'direct', form = 'binary', recl = 8)
write(2) 'C12-', '92'              ! Запись в файл 6 байт и двух null-символов
write(2) 'C16-', '99'              ! Вторая запись двоичного файла
rewind 2
read(2) ch                         ! Читаем 14 байт из двоичного файла
write(*, *) ch                     ! C12-92  C16-99
end          ! Прочитать 14 байт из неформатного файла, открытого с recl = 8, нельзя
```

Текстовый файл прямого доступа устроен так же, как и текстовый файл последовательного доступа, в котором все записи имеют одинаковую длину. Поэтому текстовый файл с равными по длине записями может быть открыт как для прямого, так и для последовательного доступа.

Пример. В текстовом файле прямого доступа каждая запись имеет поля "Фамилия - И.О. - Группа". По ошибке в некоторых записях фамилия записана со строчной буквы. Исправить ошибку в исходном файле и сформировать отчет в виде двоичного файла, содержащего исправленные записи.

```
type gru
 character(30) lastn, firstn, gru*8
end type gru
type (gru) line
integer(2) :: code, dco, cco, ut = 2, ub = 3
```

```
logical :: fl = .false.
open(ut, file='a.txt', form='formatted', access='direct', recl=68)
open(ub, file = 'b.dat', form = 'binary')
code = ichar('Z')                    ! 90 - ASCII-код для символа Z
dco = ichar('z') - code              ! dco = 32
do
 read(ut, '(2a30, a8)') line.lastn, line.firstn, line.gru          ! Или  '(3a)'
 fl = eof(ut)
 cco = ichar(line.lastn(1:1))
 if(cco > code) then                 ! Если первая буква фамилии строчная:
                                     ! Перевод строчной буквы в прописную
  line.lastn(1:1) = char(cco - dco)
  backspace ut                       ! Исправление ошибки в исходном файле
  write(ut, '(2a30,a8)') line.lastn, line.firstn, line.gru
  write(ub) line                     ! Запись в двоичный файл
 endif
 if( fl ) exit                       ! Если конец файла
enddo
close(ut)
rewind ub                            ! Контрольный вывод
do while(.not. eof(ub))
 read(ub) line
 write(*, *) line.lastn, line.firstn, line.gru
enddo
close(ub)
end
```

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').
```

Файл a.txt. Последний символ должен быть в 68-й позиции (RECL = 68).

```
Name1                    io1                    gru1
aname2                   io2                    gru2
Name3                    io3                    gru3
bname4                   io4                    gru4
Name                     io5                    gru5
cname6                   io6                    gru6
Name7                    io7                    gru7
Name8                    io8                    gru8
Name9                    io9                    gru9
dname10                  io10                   gru10
```

10.6. Удаление записей из файла прямого доступа

Способ удаления хвостовых записей из файла прямого доступа 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 выде-

лить 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:

- переместить помеченные для удаления записи в конец файла, не меняя порядока следования сохраняемых записей;
- переместиться на первую помеченную для удаления запись и выполнить оператор ENDFILE.

Запуск такой процедуры следует выполнять по мере необходимости, памятуя, что удаленные записи не могут быть восстановлены.

10.7. 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;
- программирование двоичного и неформатного В/В значительно проще, чем форматного;
- двоичные и неформатные файлы, как правило, имеют меньший размер, чем текстовые файлы с теми же данными.

Последнее обстоятельство проиллюстрируем примером:

```
real :: 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. Проще работать с последовательными устройствами. Однако 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. Каждому такому устройству В/В присваивается номер. Между номером устройства *u* и именем файла *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. В Фортране 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);
- блокировки и разблокировки (опция SHARE оператора OPEN);
- опроса (INQUIRE и EOF).

Закрывается файл в результате выполнения оператора CLOSE или при нормальном завершении программы.

Все приведенные операторы (кроме PRINT), а также функция EOF содержат в качестве одного из параметров номер устройства, к которому подсоединяется файл в результате выполнения оператора OPEN. Этот параметр при работе с внешними файлами является обязательным. В то же время каждый из работающих с файлами операторов имеет и необязательные для применения параметры. Целесообразность их применения определяется как решаемыми задачами, так и требованиями к надежности программного продукта.

Большинство из рассматриваемых в этой главе операторов были введены ранее. Однако они применялись, как правило, только с обязательными параметрами. Теперь мы выполним полное описание всех возможных операций над файлами и их выполняющих операторов. Как и ранее, для указания текущей позиции файла мы будем пользоваться понятием файлового указателя (ФУ). При описании операторов их необязательные элементы заключаются в квадратные скобки. Символ | используется для обозначения "или".

11.1. Оператор BACKSPACE

Оператор возвращает файл на одну запись назад и имеет две формы:

BACKSPACE *u*

BACKSPACE ([UNIT =] *u* [, ERR = *err*] [, IOSTAT = *iostat*])

u - выражение стандартного целого типа, задающее номер подсоединенного к файлу устройства. При отсутствии подсоединения возникнет ошибка выполнения.

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.

Оператор BACKSPACE позиционирует файл ровно на одну запись назад, кроме приведенных в табл. 11.1 случаев.

Таблица 11.1. Специальные случаи позиционирования файла оператором BACKSPACE

Случай	*Результат*
Нет предшествующей записи	Положение ФУ не меняется
Предшествующая запись - "конец файла"	ФУ позиционируется до записи "конец файла"
ФУ находится в пределах одной записи	ФУ перемещается в начало этой записи

В файлах, созданных при выводе 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 слэш (/).

Пример.

```
integer :: lunit = 10, ios
backspace 5
backspace (5)
backspace lunit
backspace (unit = lunit, err = 30, iostat = ios)
```

Замечания:

1. Если опция UNIT= опущена, 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 опцию UNIT=.

2. 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 ФУ в начало первой записи файла.

REWIND *u*

или

REWIND([UNIT =] *u* [, ERR = *err*] [, IOSTAT = *iostat*])

u - выражение стандартного целого типа, задающее номер подсоединенного к файлу устройства. При отсутствии подсоединения оператор REWIND не вызывает никаких действий.

Параметры *err* и *iostat* имеют тот же смысл, что и аналогичные параметры оператора BACKSPACE.

11.3. Оператор ENDFILE

Оператор записывает специальную запись "конец файла".

ENDFILE *u*

или

ENDFILE ([UNIT =] *u* [, ERR = *err*] [, IOSTAT = *iostat*])

u - выражение стандартного целого типа, задающее номер устройства. Если файл не открыт, то возникает ошибка выполнения.

Параметры *err* и *iostat* имеют то же смысл, что и аналогичные параметры операторов BACKSPACE и REWIND.

После выполнения записи "конец файла" ФУ 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", стираются. Это справедливо как для последовательных файлов, так и для файлов прямого 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.

```
OPEN ([UNIT =] u [,  ACCESS = access] [, ACTION=action]      &
[, BLANK = blank] [, BLOCKSIZE = blocksize]                  &
[, CARRIAGECONTROL = carriagecontrol] [, DELIM = delim]      &
[, ERR = err] [, FILE = file] [, FORM = form]                &
[, IOFOCUS = iofocus] [, IOSTAT = iostat] [, MODE = mode]    &
[, PAD = pad] [, POSITION = position] [, RECL = recl]        &
[, SHARE = share] [, STATUS = status])
```

u - выражение стандартного целого типа, задающее номер устройства, 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'). Опция ACCESS = 'APPEND' применяется при работе с последовательными файлами, открываемыми для добавления данных.

При успешном выполнении оператора OPEN с опцией 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 в файл) или 'READWRITE' (можно и читать и записывать данные).

Если *опция* ACTION не задана, 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 опции STATUS= не оказывает никакого влияния на *action*.

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.

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 задана опция FORM = 'UNFORMATTED' или FORM = 'BINARY', то опция *carriagecontrol* игнорируется.

delim - символьное выражение, задающее ограничитель для символьных данных при B/B 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. 9.8.1.2).

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 *file* не задан, то создается временный, стираемый после выполнения оператора CLOSE или после нормального

завершения программы файл. Если параметр *file* есть пробел (FILE = ' '), 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 FPS над стандартом Фортран 90.

iostat - целочисленная переменная, возвращающая 0 при отсутствии ошибок, отрицательное число, если обнаружен конец файла, или номер возникшей ошибки.

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 FPS над стандартом Фортран 90.

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
```

```
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 в 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.

recl - целочисленное выражение, задающее длину каждой записи в байтах. Этот параметр задается только в файлах прямого доступа.

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.

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.

Подсоединение файла к устройству звездочка (*) не имеет никакого действия, поскольку это устройство постоянно связанно с клавиатурой и экраном. Однако можно подсоединить файл к подсоединенным по умолчанию устройствам 0, 5 и 6.

Если в операторе OPEN использован номер устройства, подсоединенного 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, задавая в нем имя уже подсоединенного файла для изменения следующих свойств подсоединения: BLANK=, DELIM=, PAD=, ERR= и IOSTAT=. В таких случаях опция POSITION = 'APPEND' игнорируется, но опция POSITION = 'REWIND' вызывает перемещение на начало файла.

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')
```

11.5. Оператор CLOSE

Оператор отсоединяет файл от устройства В/В и закрывает это устройство.

CLOSE ([UNIT =] *u* [, ERR = *err*] [, IOSTAT = *iostat*] [, STATUS = *status*])

u - целочисленное выражение, задающее номер устройства. Если к устройству не был подсоединен файл, то никакой ошибки не возникает.

Параметры *err* и *iostat* имеют тот же смысл, что и аналогичные параметры операторов BACKSPACE и REWIND.

status - символьное выражение, вычисляемое со значениями 'KEEP' или 'DELETE'. Для временных (*scratch*) файлов по умолчанию принимается статус 'DELETE'. Задание STATUS = 'KEEP' для временных файлов вызывает ошибку выполнения. Для других видов файлов по умолчанию принимается статус 'KEEP'.

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 соответ-

ственно 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. Передача данных выполняется до тех пор, пока не выполнены все операции ввода, либо пока не достигнут конец файла, либо пока не возникла ошибка ввода. В случае ввода под управлением списка ввод прекращается при обнаружении в поле ввода слэша (/).

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 опция UNIT= опущена, то параметр *u* должен быть первым параметром оператора. Если опущены опции FMT= или NML=, то параметры *fmt* или *nml* должны быть вторыми параметрами оператора. В противном случае параметры могут появляться в произвольном порядке.

u - номер устройства.

При вводе из внутреннего файла номером устройства является имя строки, подстроки или символьного массива (перечисленные переменные могут быть элементами записи). При вводе из внешнего файла номером устройства является целочисленное выражение. Номер устройства может быть задан 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, то есть после выполнения каждого оператора ввода ФУ 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 параметр *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 параметры *err* или *iostat*, то чтение после записи "конец файла" приведет к ошибке исполнения.

eor - метка оператора того же блока видимости, в котором размещен оператор OPEN. Если опция EOR= задана, то также должна быть задана и опция ADVANCE='NO'. Если опция EOR= задана, выполнение оператора ввода прекращается при достижении конца записи (если ранее не было иной ошибки). Если опция EOR= опущена, то при достижении конца записи возникает ошибка, которую можно обработать опцией IOSTAT=.

err - метка исполняемого оператора. При возникновении ошибки В/В управление передается на оператор, имеющий метку *err*.

iostat - целочисленная переменная, возвращающая 0 при отсутствии ошибок, -1, если обнаружен конец файла или номер возникшей ошибки. Состояние ошибки возникает, например, если обнаружен конец записи при непродвигающемся вводе.

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). Если параметр *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. По умолчанию значение *rec* равно номеру текущей записи файла. И если при вводе из прямого файла параметр отсутствует, то будет введена текущая запись файла.

size - целочисленная переменная стандартного целого типа, возвращающая число переданных при выполнении форматного ввода полей с данными. Добавляемые в результате выполнения опции PAD='YES' про-

белы не засчитываются. Опция SIZE= может быть задана только при задании опции 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)
enddo
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, типов, 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 WRITE

Оператор передает данные из списка вывода в файл, подсоединенный к устройству *u*.

WRITE ([UNIT =] *u* [, [[FMT =] *fmt*] | [NML=] *nml*] &
[, ADVANCE = *advance*] [, ERR = *err*] &
[, IOSTAT = *iostat*] [, REC = *rec*]) [*iolist*]

u - номер устройства. При выводе во внутренний файл номером устройства является имя строки, подстроки или символьного массива (перечисленные переменные могут быть элементами записи). При выводе во внешний файл номером устройства является целочисленное выражение. Вывод будет выполняться на экран, если в качестве номера 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 использо-

вано 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, то есть после выполнения каждого оператора вывода проставляются символы конца записи и ФУ 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. По умолчанию значение *rec* равно номеру текущей записи файла. И если при выводе в прямой файл параметр отсутствует, то будет изменена текущая запись файла.

iolist - список вывода, содержащий выражения, результаты которых должны быть переданы в файл.

При записи в последовательный файл все записи, расположенные после введенной, удаляются и ФУ позиционируется на конец файла. Таким образом, после вывода в последовательный файл необходимо применить BACKSPACE или REWIND для выполнения оператора READ.

11.8. Оператор PRINT

Оператор выводит данные на экран (устройство *).

PRINT * | *fmt* [, *iolist*]

где *fmt* - спецификатор формата; *iolist* - список вывода. Если звездочка замещает *fmt*, то вывод управляется списком *iolist*.

11.9. Оператор INQUIRE

Оператор возвращает свойства устройства или внешнего файла.

```
INQUIRE ([UNIT =] u | FILE = file | IOLENGTH = iolength
[, ACCESS = access] [, ACTION = action] [, BINARY = binary]                    &
[, BLANK = blank] [, BLOCKSIZE = blocksize]                                    &
[, CARRIAGECONTROL = carriagecontrol] [, 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] [, PAD = pad] [, POSITION = position]                      &
[, READ = read] [, READWRITE = readwrite] [, RECL = recl]                      &
[, SEQUENTIAL = seq][, SHARE = share]                                          &
[, UNFORMATTED = unformatted] [, WRITE = write]) [iolist]
```

u - звездочка (*) или целочисленное выражение, задающие номер устройства, о котором необходимо получить информацию. Если задано UNIT = *, то нельзя включать опцию NUMBER.

file - символьное выражение, задающее имя файла, информацию о котором возвращает оператор INQUIRE.

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 параметр *file*, то выполняется опрос файла.

iolength - переменная стандарного целого типа, возвращающая размер списка вывода. Эта форма оператора INQUIRE включает только опцию IOLENGTH= и список вывода *iolist*. Все другие опции 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 параметра RECL= оператора OPEN. Затем данные можно передать в открытый неформатный файл прямого доступа.

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.

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.

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 Windows-приложениями.

iostat - переменная стандарного целого типа. Возвращает 0 при отсутствии ошибок, отрицательное число, если обнаружен 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 .FALSE., если файл не открыт, и возвращает .TRUE. в противном случае.

nextrec - переменная стандарного целого типа. Возвращает номер следующей записи в файле прямого доступа. Номер первой записи файла равен единице.

num - переменная стандарного 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 опцию NUMBER=.

opened - логическая переменная, возвращающая при опросе устройства .TRUE., если какой-либо файл подсоединен к устройству, и возвращающая .FALSE. в противном случае. При опросе файла возвращает .TRUE., если файл подсоединен к какому-либо устройству, и возвращает .FALSE. в противном случае.

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 ФУ расположен в конечной точке файла 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 стандарного 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.

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 *unit* 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
```

11.10. 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:

EOF(*u*)

u - выражение стандартного целого типа, задающее номер устройства.

12. Вывод графических данных

При работе с графическими данными решают задачи:

- создания изображения;
- редактирования изображения;
- выбора структур данных для хранения графической информации в оперативной и внешней памяти (внутреннее и внешнее представление данных);
- ввода графических данных и воспроизведения по прочитанным данным изображения и другие.

Создаваемые графические изображения с целью их повторного использования записываются в файлы. Часто файлы с графическими данными содержат и дополнительную неграфическую информацию. Например, в пакетах анимации в файл добавляются алгоритмы изменения изображения и его отдельных частей.

Графические данные могут отображать двумерные, трехмерные и в принципе любые *n*-мерные объекты. В практических приложениях используются двумерные и трехмерные модели.

Независимо от того, какие объекты хранятся в графических данных (двумерные или трехмерные), вывод изображения осуществляется на плоскость (экран монитора, принтер...).

В настоящей главе мы рассмотрим только один аспект работы с графическими данными: вывод изображения на экран дисплея. Причем будут рассмотрены только графические процедуры, применяемые в проектах со стандартной графикой и проектах QuickWin. Проект со стандартной графикой является однооконным, а проект QuickWin - многооконным графическим проектом, в котором также можно реализовать некоторые возможности Win32 Application Programming Interface (API). Однако этим графические возможности FPS не ограничиваются. В FPS могут быть созданы проекты с любыми свойствами Windows-приложений.

Используя технику Windows-программирования, в FPS можно:

- снабдить программу Windows графическим пользовательским интерфейсом;
- получить доступ к Windows-графическим процедурам;
- получить доступ к службе управления системой нижнего уровня: регистрам, функциям виртуальной и разделяемой памяти и к службе управления системой верхнего уровня, например к сетевым функциям.

Доступ к Win32 API и создание Windows-приложений становятся возможными после включения в программу модуля MSFWIN.

Помимо этого, в FPS можно обеспечить доступ к библиотеке OpenGL, содержащей графические функции создания трехмерных изображений и анимаций.

12.1. Графический дисплей

В современных ЭВМ используются растровые дисплеи (существуют также векторные дисплеи). Графические возможности растрового дисплея определяются размером экрана, зерном экрана, частотой регенерации и разрешением.

Размер экрана дисплея может варьироваться от 14 до 21 и более дюймов.

Вывод изображения выполняется в результате подсветки электронным лучом отдельных регулярно расположенных точек экрана. Такая точка является наименьшим элементом графической информации и называется *видеопикселем* или просто *пикселем*.

Число таких точек по горизонтали и вертикали экрана определяет его *разрешение*. Стандартными являются разрешения 640 * 480 (первая цифра указывает на число точек по горизонтали, вторая - по вертикали), 800 * 600, 1024 * 768, 1280 * 1024 и более высокие разрешения. На одном дисплее могут быть установлены разные разрешения. Управление разрешением выполняется графическим адаптером.

Луч последовательно "пробегает" все точки экрана, выполняя заданную программой для каждой точки подсветку. Затем цикл обхода повторяется. Полный цикл обхода всех точек экрана приводит к воспроизведению изображения и называется *регенерацией изображения*. Число регенераций в секунду называется *частотой регенерации*. Низкая частота регенерации приводит к заметному для человеческого глаза миганию изображения. Хорошим показателем можно считать частоту регенерации не менее 60 Гц.

Расстояние между соседними точками экрана определяет его *зерно*. Размеры зерна варьируются от 0.31 мм до 0.21 мм и менее. Понятно, что чем меньше зерно, тем выше качество изображения.

12.2. Растровое изображение

В графике экран растрового дисплея представляется в виде прямоугольной сетки на дискретной плоскости с шагом по осям *x* и *y*, равным единице. Такая модель называется *растровой плоскостью* или *растром* (рис. 12.1, *а*). Массив прямоугольных ячеек плоскости называется *растровым массивом*.

Каждый квадрат сетки соответствует одному пикселю экрана. Точка инициализации пикселя находится в центре квадрата сетки. Инициализации точки растра с координатами (i, j) соответствует закраска квадрата сетки, в центре которого эта точка расположена (рис. 12.1, *б*).

Растровое изображение создается путем закраски ячеек растрового массива в тот или иной цвет. Растровое изображение можно сравнить с изображением на листе клетчатой бумаги, получаемым в результате закраски отдельных клеточек листа.

Каждому пикселю растра могут быть независимым образом заданы цвет, интенсивность и другие характеристики.

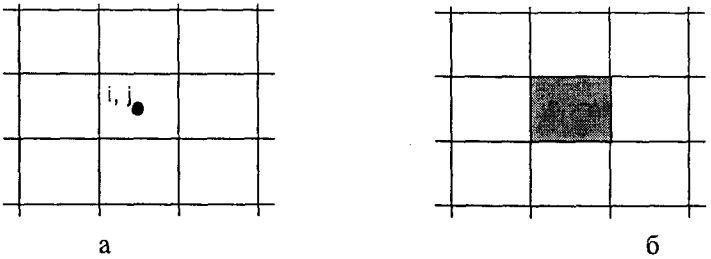

Рис. 12.1. Модель растрового экрана:
а - растровая плоскость; б - инициализация точки растра

При создании черно-белых изображений для хранения информации о цвете одного пикселя достаточно 1 бита памяти, записывая в соответствующую ячейку памяти 1, если элемент изображения закрашен в черный цвет, и 0 - если в белый.

Цветное изображение создается комбинацией базовых цветов, в качестве которых могут быть использованы, например, красный - зеленый - синий (RGB) или голубой - пурпурный - желтый - черный (CMYK). Задавая различную интенсивность при наложении базовых цветов, можно воспроизвести все различимые человеческим глазом цвета и оттенки. При работе с палитрой, содержащей 256 цветов, для хранения информации о цвете одного пикселя потребуется уже 1 байт (8 битов) памяти.

В системе цветов RGB каждый пиксель содержит 3 точки, каждая из которых "отвечает" за свой цвет: красный, зеленый, синий. Комбинируя эти цвета и их интенсивность, можно управлять цветом каждого пикселя и, следовательно, всего изображения. Человеческий глаз не в состоянии различить отдельные точки пикселя в силу их маленького размера и воспринимает как целое получаемое в результате наложения цветов изображение.

Из сказанного следует, что для построения изображения в принципе достаточно иметь одну функцию, позволяющую изменять цвет любого пикселя растра. В FPS такая функция имеет имя SETPIXELRGB.

12.3. Видеоадаптер

Данные, подлежащие воспроизведению на экране монитора, формируются программой, функционирующей на центральном процессоре ЭВМ. Далее они передаются видеоадаптеру, который размещает эти данные в видеопамяти, преобразовывает и передает устройству управления лучом - электронно-лучевой трубке.

Таким образом, *видеоадаптер* является устройством сопряжения центрального процессора ЭВМ и устройства отображения.

Видеоадаптеры могут обеспечить разные режимы отображения данных, которые разделяются на *текстовые* и *графические.* Причем, как правило, видеоадаптер поддерживает несколько графических режимов. Режим отображения данных на экране называется *видеорежимом.* В FPS

видеорежим устанавливается при задании конфигурации видеоокна функцией SETWINDOWCONFIG.

Различные видеорежимы отличаются количеством выводимых на экран данных и цветов. Единицами измерения количества выводимых данных являются: в текстовом режиме - символ (литера), в графическом - видеопиксель (или просто пиксель). Например, видеоадаптер SVGA позволяет установить текстовый режим, в котором на экран выводится 37 строк по 100 символов в каждой (100 * 37), и графический режим, обеспечивающий разрешение 800 * 600 пикселей. Возможны, разумеется, и иные режимы работы видеоадаптера.

В *текстовом* режиме 100 * 37 символ имеет размер 8 * 8 пикселей. Матрица 8 * 8 пикселей, используемая для отображения символа на экране, называется *знакоместом*. Символ передается в видеоадаптер в виде двухбайтовой последовательности. Первый байт - код символа. Второй - его атрибуты: цвет символа и фона.

В *графическом* режиме отображается два типа объектов: литеры (символы) и геометрические объекты.

При отображении литер используются шаблоны (шрифты). В видеоадаптер для отображения строки литер следует передать (ASCII) коды литер, атрибуты строки (высота, угол наклона, цвет и другие), а также имя шрифта (имя файла, в котором размещены формы литер). При отображении литеры будет, во-первых, найдена форма литеры в указанном шрифте, выполнено преобразование (масштабирование, изменение угла наклона) и отображение преобразованной формы литеры на экране монитора в соответствии с ее атрибутами.

Построение геометрических объектов выполняется по заданным точкам. Для геометрических объектов так же, как и для литер, могут быть определены атрибуты: цвет, тип и толщина линии и другие.

12.4. Видеоокно и окна вывода

В FPS в проектах с графикой вывод и текстовых и графических данных выполняется в *видеоокно*, параметры которого устанавливаются функцией SETWINDOWCONFIG. В проекте со стандартной графикой получаемое в начальный момент изображение отображается на полном экране без меню и вертикальной и горизонтальной полос прокрутки экрана. Затем видеоокно получает необходимые для него меню и полосы прокрутки; размеры и положение видеоокна могут быть изменены стандартным для Windows-приложений способом.

В QuickWin каждое создаваемое (дочернее) видеоокно всегда располагается внутри обрамляющего окна. Размеры и положение обрамляющего и любого дочернего видеоокна могут быть изменены не только пользователем, но и из программы (разд. 12.15).

По умолчанию открываемое видеоокно ассоциируется при выводе с устройствами *, 0 и 6, а при вводе - с устройствами * и 5. В приложениях QuickWin оператором OPEN (разд. 11.5) видеоокно можно подсоединить и к нестандартному устройству. Вывод текстовых данных в видеоокно

выполняется операторами PRINT, WRITE и подпрограммой OUTTEXT, чтение - оператором READ.

Внутри видеоокна в FPS могут быть созданы прямоугольные области вывода, называемые *окнами вывода*. По умолчанию окном вывода является все видеоокно, на которое и направляется как графический, так и текстовый вывод. С этим окном связана физическая система координат (разд. 12.6). Единицей измерения значений текстовых координат является знакоместо, а графических - пиксель. Затем программист внутри видеоокна может выделить окна для вывода текстовых и графических данных. Каждое такое окно имеет свою систему координат. Причем для вывода графических данных в FPS могут быть созданы окна с вещественной системой координат, что существенно упрощает работу с реальными физическими объектами. Число создаваемых окон вывода произвольно.

Замечание. Далее, как правило, мы будем употреблять слово "окно" вместо более громоздкого слова "видеоокно".

Рассматриваемые ниже графические процедуры относятся к процедурам стандартной графики. Работать с ними можно в проектах QuickWin или Standard Graphics. В каждый программный компонент, где вызываются графические процедуры, необходимо включить модуль MSFLIB (USE MSFLIB). Этот модуль содержит интерфейсы к графическим процедурам, необходимые для работы с графикой именованные константы и определения производных типов данных. Например, интерфейс к процедуре очистки экрана выглядит так:

```
INTERFACE
 SUBROUTINE clearscreen[C,ALIAS:"__FQclearscreen"](area)
  INTEGER(2) area
 END SUBROUTINE
END INTERFACE
```

12.5. Задание конфигурации видеоокна

Установка конфигурации видеоокна не является обязательной процедурой, но при необходимости может быть выполнена функцией

SETWINDOWCONFIG (*wc*)

wc - переменная производного типа *windowconfig*, содержащая параметры окна. Тип *windowconfig* определен в модуле MSFLIB:

```
TYPE windowconfig
  INTEGER(2) numxpixels      ! число пикселей по оси x
  INTEGER(2) numypixels      ! число пикселей по оси y
  INTEGER(2) numtextcols     ! число доступных столбцов текста
  INTEGER(2) numtextrows     ! число доступных рядов текста
  INTEGER(2) numcolors       ! число индексов цвета
  INTEGER(4) fontsize        ! Размер устанавливаемого по умолчанию шрифта.
                             ! Принимает значение QWIN$EXTENDFONT
                             ! в случае многобитовых символов.
                             ! С многобитовыми шрифтами используется
                             ! параметр extendfontsize
```

```
  CHARACTER(80) title              ! Заголовок окна (С-строка)
  INTEGER(2) bitsperpixel          ! Число бит на пиксель
! Следующие 3 параметра введены для поддержки многобитовых символов
! (например, символов японского языка). Параметр fontsize при переходе
! к многобитовым шрифтам должен иметь значение QWIN$EXTENDFONT
  CHARACTER(32) extendfontname                ! Любой непропорциональный шрифт
  INTEGER(4) extendfontsize        ! Принимает то же значение, что и
                                   ! fontsize, но используется при работе
                                   ! с многобитовыми шрифтами.
  INTEGER(4) extendfontattributes ! Атрибуты многобитовых шрифтов (жирный или курсив)
END TYPE windowconfig
```

Функция возвращает значение стандартного логического типа, равное .TRUE., если конфигурация установлена, и .FALSE. - в противном случае.

Если некоторым числовым компонентам *windowconfig* задать значение −1, то функция SETWINDOWCONFIG установит для них наивысшее возможное в вашей системе разрешение, сохраняя значения других, отличных от -1 компонентов.

Можно задать реальные значения влияющих на размер окна компонентов: число пикселей по *x* и *y*, число столбцов и рядов текста и размер шрифта - и затем вызвать SETWINDOWCONFIG.

Можно вообще не использовать SETWINDOWCONFIG. В этом случае окно выберет наилучшее из возможных разрешений и размер шрифта 8 * 16. Число доступных цветов зависит от видеоадаптера.

Если используется SETWINDOWCONFIG, то следует задать значения каждого компонента *wc* (-1 или свое собственное значение и СИ-строку для заголовка окна). Если же используется SETWINDOWCONFIG и часть компонентов предварительно не определены, то это может привести к тому, что эти компоненты примут нежелательные значения.

Если задана конфигурация, которая не может быть установлена, то SETWINDOWCONFIG возвращает .FALSE. и вычисляет значения параметров, с которыми возможна работа и которые близки к заданным значениям. Для установки окна с вычисленными параметрами необходимо выполнить второй вызов SETWINDOWCONFIG, например:

```
status = setwindowconfig(wc)
if (.not. status) status = setwindowconfig(wc)
```

Если заданы значения всех влияющих на размер окна параметров: *numxpixels*, *numypixel*, *numtextcols* и *numtextrows*, то размер шрифта вычисляется по этим значениям. По умолчанию устанавливается шрифт Courier New с размером 8 * 16.

В случае стандартного графического приложения, если задано разрешение, совпадающее со значениями, задаваемыми графическим адаптером (или -1 для параметров, влияющих на размер окна), то приложение начинает выполняться на всем экране. В противном случае приложение начинает выполняться в видеоокне. Для переключений полный экран - окно и окно - полный экран можно использовать ALT+ENTER.

По умолчанию курсор в видеоокне отсутствует. Однако после вызова функции DISPLAYCURSOR($GCURSORON) он становится видимым.

Пример.

```
use msflib
type (windowconfig) wc
logical res /.false./
wc.numxpixels = 800
wc.numypixels = 600
wc.numtextcols = -1
wc.numtextrows = -1
wc.numcolors = -1
wc.title = "Первое графическое окно"c
wc.fontsize = -1
res = setwindowconfig(wc)
if (.not.res) res = setwindowconfig(wc)
```

Значения параметров окна возвращаются функцией

GETWINDOWCONFIG (*wc*)

wc - переменная типа windowconfig, содержащая конфигурацию активного дочернего графического окна.

Функция возвращает значение стандартного логического типа, равное .TRUE., в случае успешного выполнения, и .FALSE. - в противном случае (если нет активного дочернего окна).

Если для задания параметров окна функция SETWINDOWCONFIG не была использована, то GETWINDOWCONFIG возвращает *wc* с устанавливаемыми по умолчанию значениями компонентов. Число возвращаемых цветов зависит от графического адаптера. Заголовок окна - "Graphic1". Все эти значения могут быть изменены функцией SETWINDOWCONFIG.

Поле *bitsperpixel* в типе *windowconfig* доступно только для чтения, в то время как значения всех иных полей могут быть изменены.

Пример. Вывести параметры графического окна.

```
use msflib
type(windowconfig) wc
logical res /.false./
wc.numxpixels = -1
wc.numypixels = -1
wc.numtextcols = -1
wc.numtextrows = -1
wc.numcolors = -1
wc.title= "Первое графическое окно"c
wc.fontsize = -1
res = setwindowconfig(wc)
res = getwindowconfig(wc)
print *, 'Resolution:', wc.numxpixels, ' *', wc.numypixels          ! 800*600
print *, 'Text screen:', wc.numtextcols, ' *', wc.numtextrows       ! 100*37
print *, 'Numcolors:', wc.numcolors                                ! 16
print *, 'Fontsize:', wc.fontsize    ! 524304 (8*16)
print *, 'Bit per pixel:', wc.bitsperpixel                         ! 1
end
```

12.6. Системы графических координат. Окно вывода

Растровая плоскость расположена в *физической системе координат*, определяемой техническими средствами. Начало физической системы координат - левый верхний угол видеоокна. Ось абсцисс направлена слева направо; ось ординат - сверху вниз (рис. 12.2). Физические координаты целочисленны. Наименьшие координаты пикселя $x_{min} = 0$ и $y_{min} = 0$. Наибольшие координаты определяются установленным разрешением. Так, при разрешении 800 * 600 наибольшие координаты пикселя $x_{max} = 799$ и $y_{max} = 599$. Используемый при задании координат тип данных - INTEGER(2).

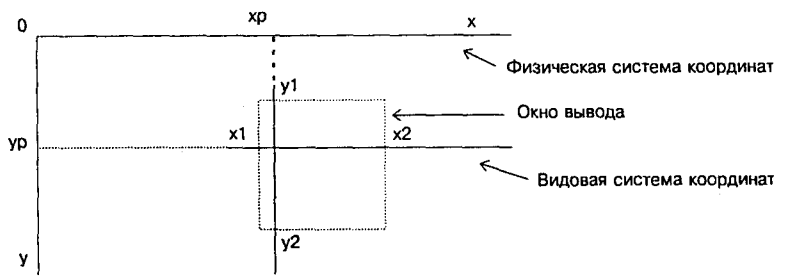

Рис. 12.2. Системы графических координат

Точка начала отсчета координат пикселей может быть перемещена в любую точку видеоокна *xp, yp*. Эта процедура называется *заданием видового порта*. Так, после перемещения точки отсчета в центр видеоокна минимальные координаты пикселя при разрешении 800 * 600 будут равны (-400, -300), а максимальные - (399, 299).

Начало координат видового порта задается подпрограммой

CALL SETVIEWORG (*xp, yp, t*)

xp, yp - выражения типа INTEGER(2), задающие начало координат нового видового порта в физической системе координат (рис. 12.2).

t - переменная типа *xycoord*, содержащая значения *xcoord, ycoord* начала координат предыдущего видового порта. Тип *xycoord* определен в модуле MSFLIB:

```
type xycoord
 integer(2) xcoord          ! x-координата
 integer(2) ycoord          ! y-координата
end type xycoord
```

Подпрограмма SETVIEWORG перемещает начало координат (0, 0) в точку физической системы координат (*xp, yp*).

Получаемая таким образом система координат называется *видовой системой координат*. Задаваемые в видовой системе координаты называются *видовыми координатами*.

По умолчанию при инициализации графического режима начало координат текущего видового порта совпадает с началом физической системы координат.

Пример. Нарисовать отрезок прямой в физической системе координат. Координаты отрезка (40, 20) и (400, 200).

```
use msflib                          ! Режим 800*600, 16 цветов
integer(2) :: status2, xa = 40, ya = 20, xb = 400, yb = 200
integer(4) :: status4
type (xycoord) xy
status4 = setbkcolor(8)             ! Цвет фона - серый
call clearscreen($gclearscreen)     ! Заливка видеоокна цветом фона
status2  = setcolor(9)              ! Текущий цвет - ярко-синий
call moveto(xa, ya, xy)             ! Перемещение в точку (xa, ya)
status2 = lineto(xb, yb)            ! Вывод отрезка
end
```

Если бы мы работали в видовой системе координат, начало которой расположено, например, в центре видеоокна, то та же самая линия имела координаты (-360, -280) и (0, -100) и для ее вывода должны быть выполнены операторы

```
call setvieworg (400_2, 300_2, xy)
call moveto(-360_2, -280_2, xy)
status2 = lineto(0_2, -100_2)
```

По умолчанию областью вывода графических данных является вся растровая плоскость (все видеоокно). Однако полезно иметь возможность выделять в видеоокне разные области вывода данных. В графических системах выделяются прямоугольные области вывода, называемые *окнами вывода*. Выделение окна вывода графических данных в FPS реализуется подпрограммой

CALL SETCLIPRGN (*x*1, *y*1, *x*2, *y*2)

*x*1, *y*1 - координаты верхнего левого угла окна вывода (см. рис. 12.2).
*x*2, *y*2 - координаты нижнего правого угла окна вывода.

Тип параметров *x*1, *y*1, *x*2, *y*2 - INTEGER(2).

Функция ограничивает область вывода данных. Физические координаты (*x*1, *y*1) и (*x*2, *y*2) вершин окна задаются относительно начала координат текущего видового порта. Графические объекты или их части, выходящие за границы окна вывода, не отображаются.

Замечание. Функция SETCLIPRGN ограничивает вывод только графических элементов, перечень которых дан в разд. 12.10. Ограничение области вывода текста при помощи OUTTEXT, WRITE и PRINT выполняется подпрограммой SETTEXTWINDOW.

Видовой порт (видовую систему координат) и одновременно окно вывода можно задать одной подпрограммой

CALL SETVIEWPORT (*x*1, *y*1, *x*2, *y*2)

*x*1, *y*1 - координаты верхнего левого угла окна вывода.
*x*2, *y*2 - координаты нижнего правого угла окна вывода.

Тип параметров *x*1, *y*1, *x*2, *y*2 - INTEGER(2).

Подпрограмма переопределяет графический видовой порт и задает окно вывода так же, как это выполняется в результате задания окна вывода подпрограммой SETCLIPRGN и последующей установки начала координат видового порта в левый верхний угол окна. То есть начало видовой системы координат после вызова

```
call setviewport ( x1, y1, x2, y2 )
```

будет совпадать с верхним левым углом окна вывода.

Координаты (*x*1, *y*1) и (*x*2, *y*2) задаются в физической системе координат. Все последующие преобразования посредством функции SETWINDOW выполняются в видовой, а не в физической системе координат.

В рассмотренных системах мы имели дело с целочисленными координатами, диапазон изменения которых определялся разрешением дисплея и заданным окном. В FPS, однако, есть возможность задать окно вывода, связав его с *оконной системой координат* (ОСК), в которой используются вещественные координаты. Задаваемые в оконной системе координаты называются *оконными координатами*. ОСК может совпадать с *мировой системой координат*, то есть с системой, в которой мы работаем с реальными объектами.

Окно вывода, в котором можно работать с вещественными координатами, задается в видовой системе координат функцией

SETWINDOW (*finvert*, *wx*1, *wy*1, *wx*2, *wy*2)

finvert - параметр типа LOGICAL(2), определяющий направление координат. Если *finvert* равен .TRUE., то ось *y* увеличивается от нижней границы видового порта к верхней. Если *finvert* равен .FALSE., то ось *y* увеличивается от верхней границы видового порта к нижней, то есть так, как и в видовой системе координат.

*wx*1, *wy*1 - верхний левый угол окна в ОСК.

*wx*2, *wy*2 - нижний правый угол окна в ОСК.

Тип параметров *wx*1, *wy*1, *wx*2, *wy*2 - REAL(8). Тип функции - INTEGER(2). Функция возвращает отличное от нуля значение при успешном открытии окна и 0 в противном случае.

Функция SETWINDOW определяет ОСК, в которой координаты выводимых объектов ограничены значениями параметров *wx*1, *wy*1, *wx*2, *wy*2. В общем случае начало координат ОСК может находиться за пределами окна вывода.

В ОСК могут быть использованы для вывода графических элементов только функции, имена которых завершаются символами _W (например, ARC_W, RECTANGLE_W или LINETO_W).

Задание ОСК выполняется относительно текущего видового порта.

Пример. Построить график функции $y = sin(x)$ на отрезке от $-\pi$ до π.

Пусть размеры видеоокна по осям *x* и *y* равны соответственно XE и YE. Используем для вывода графика расположенный в центре видеоокна

видовой порт, ширина и высота которого равны соответственно XE/2 и YE/2. В этом видовом порте зададим оконную систему координат, начало которой расположено в центре видеоокна, а ось *y* направлена снизу вверх (рис. 12.3)

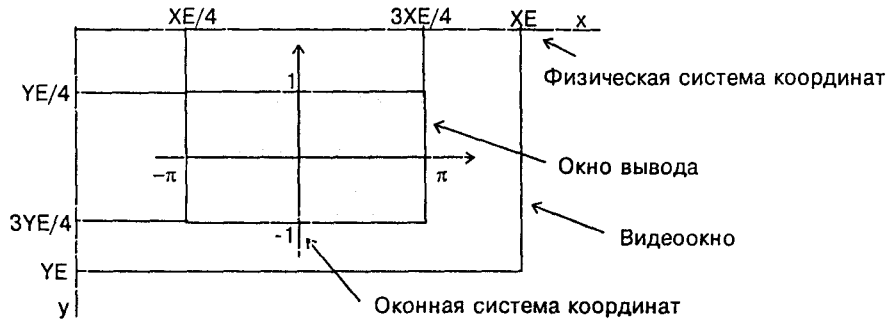

Рис. 12.3. Оконная система координат для графика y = sin(x)

```
! Рисуем график y = sin(x)
use msflib                          ! Режим 800*600, 16 цветов
integer(2) status2, XE, YE
integer(4) status4
real(8) dx, x, y
logical(2) :: finv = .true., res
real(8), parameter :: pi = 3.14159265_8
type (windowconfig) wc
status4 = setbkcolor(1)             ! Цвет фона - синий
call clearscreen($gclearscreen)     ! Заливка экрана цветом фона
res = getwindowconfig (wc)
XE = wc.numxpixels                  ! numxpixels - число пикселей по оси x
YE = wc.numypixels                  ! numxpixels - 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, YE/4, 3*XE/4, 3*YE/4)
! Оконная система координат (ОСК)
status2 =  setwindow (finv, -pi, dble(-1), pi, dble(1))
status2  = setcolor(10)             ! График светло-зеленым цветом
dx = pi / dble(XE/2)                ! Шаг по оси x
do x = -pi, pi, dx                  ! Изменение x в ОСК
 y = dsin(x)                        ! Значение y в ОСК
 status2 = setpixel_w(x, y)         ! Вывод пикселя в ОСК
enddo
contains
 subroutine axis ( )                ! Вывод осей координат
  type (xycoord) xy
  status2  = setcolor(7)            ! Оси координат - белым цветом
  call moveto(int2(XE/4 - 10), int2(YE/2), xy)
  status2 = lineto(3*XE/4 + 10, YE/2)          ! Ось x
  call moveto(int2(XE/2), int2(YE/4 - 10), xy)
  status2 = lineto(XE/2, 3*YE/4 + 10)          ! Ось y
 end subroutine
end
```

12.7. Очистка и заполнение экрана цветом фона

Видеоокно, текущий видовой порт и текущее текстовое окно вывода очищаются после применения подпрограммы

CALL CLEARSCREEN (*area*)

area - определенная в модуле MSFLIB константа, указывающая на тип очищаемой области и принимающая значения:

$GCLEARSCREEN - очистка всего видеоокна;
$GVIEWPORT - очистка текущего видового порта;
$GWINDOW - очистка текущего текстового окна, заданного посредством SETTEXTWINDOW.

Помимо очистки заданная область заполняется цветом текущего фона.

Пример. Залить видеоокно белым цветом, изобразить в его центре прямоугольник фиолетового цвета, в центре прямоугольника задать текстовое окно белого цвета, в котором темно-серыми буквами написать 'TOP'

```
use msflib                          ! Режим 800*600, 16 цветов
integer(2) status2                  ! и 100*37 для текста
integer(4) status4
type (rccoord) rc
status4 = setbkcolor(15)            ! Белый фон
call clearscreen($gclearscareen)    ! Заливка текстового окна
status2 = setcolor(5)               ! Фиолетовый цвет
status2 = rectangle($gfillinterior, 200, 100, 600, 500)
call settextwindow(15, 40, 25, 60) ! Задание текстового окна
call clearscreen($gwindow)          ! Заливка текстового окна
status2 = settextcolor(8)           ! Темно-серый цвет
call settextposition(6, 10, rc)
call outtext('TOP')
end
```

Замечание. Текстовое окно всегда задается относительно верхнего левого угла видеоокна. Позиция текста задается относительно верхнего левого угла текстового окна.

12.8. Управление цветом

12.8.1. Система цветов RGB. Цветовая палитра

Количество доступных для воспроизведения цветов определяется, с одной стороны, возможностями дисплея, видеоадаптера, используемой системой цветов, а с другой - установленным видеорежимом.

В ПЭВМ преимущественно используется система цветов RGB, в которой любой цвет получается в результате изменения интенсивности и смешения базовых цветов: красного, зеленого и синего. Конструктивно такая возможность обеспечивается устройством электронно-лучевой трубки, в которой каждый видеопиксель содержит 3 точки: красную, зеленую и синюю. Подсвечивая с различной интенсивностью эти точки, современные видеоадаптеры и дисплеи могут воспроизвести миллионы цветов

и оттенков. Однако в большинстве приложений можно обойтись 256 или 16 цветами.

При загрузке видеорежима компьютер создает *индексированную палитру цветов*, то есть такой набор цветов, в котором каждому цвету (его значению) присвоен свой номер. Число цветов в палитре можно определить после обращения к функции GETWINDOWCONFIG.

Цветовая палитра устанавливает соответствие между *значением цвета* в системе цветов RGB и номером цвета в палитре. Поясним понятие "значение цвета". Для описания каждого цвета RGB используется 3 байта (24 RGB-бита). Каждый байт хранит интенсивность красного, зеленого и синего цветов, например:

синий байт	*зеленый байт*	*красный байт*
bbbbbbbb	gggggggg	rrrrrrrr

Синий цвет наибольшей интенсивности получается, если в RGB-байтах установлено значение #FF0000 (шестнадцатеричное FF соответствует двоичному коду 11111111). Наибольшей интенсивности зеленого цвета соответствует значение #00FF00, а красного - #0000FF. При наложении трех базовых цветов максимальной интенсивности (значение #FFFFFF) получится ярко-белый цвет. Всего на трех RGB-байтах может быть задано 256 * 256 * 256 = 16'777'216 различных цветов и оттенков.

Значением цвета называется число, содержащееся в 24 RGB-битах, то есть в битах задания цвета. В табл. 12.1 приведены значения цветов для 16 цветовой палитры RGB.

Таблица 12.1. RGB значения цветов

Цвет	*RGB-значение*	*Цвет*	*RGB-значение*
Черный	#000000	Ярко-белый	#FFFFFF
Светло-красный	#000080	Красный	#0000FF
Светло-зеленый	#008000	Зеленый	#00FF00
Светло-желтый	#008080	Желтый	#00FFFF
Светло-синий	#800000	Синий	#FF0000
Светло-фиолетовый	#800080	Фиолетовый	#FF00FF
Светло-бирюзовый	#808000	Бирюзовый	#FFFF00
Светло-серый	#808080	Серый	#C0C0C0

Значение цвета типа INTEGER(4) по его известным RGB-компонентам возвращает функция

RGBTOINTEGER (*red*, *green*, *blue*)

red - параметр типа INTEGER(4), задающий значение интенсивности красного компонента цвета.

green - параметр типа INTEGER(4), задающий значение интенсивности зеленого компонента цвета.

blue - параметр типа INTEGER(4), задающий значение интенсивности синего компонента цвета.

Значения параметров *red*, *green* и *blue* находятся в диапазоне от 0 до 255.

Пример. Нарисовать квадрат фиолетового цвета на бирюзовом фоне.

```
use msflib                          ! Режим 800*600, 16 цветов
integer(2) st2
integer(4) ctm, st4
ctm = rgbtointeger(0, 255, 255)     ! Значение бирюзового цвета
st4 = setbkcolorrgb(ctm)            ! Цвет фона
call clearscreen($gclearscreen)     ! Заполнение экрана цветом фона
ctm = rgbtointeger(200, 0, 200)     ! Оттенок фиолетового цвета
st4 = setcolorrgb(ctm)              ! Цвет вывода графических данных
st2 = rectangle($gfillinterior, 300, 200, 500, 400)
end
```

Подпрограмма

CALL INTEGERTORGB (*rgb*, *red*, *green*, *blue*)

по значению цвета *rgb* типа INTEGER(4) возвращает значения его RGB-компонентов: *red*, *green*, *blue*. Тип возвращаемых значений - INTEGER(4). Например:

```
use msflib
integer(4) r, g, b
call integertorgb(13107400, r, g, b)
print *, r, g, b                    !      200      0      200
end
```

12.8.2. Цветовая палитра VGA

В режиме VGA с 16 цветами и разрешением 640 * 480 по умолчанию устанавливается приведенная в табл. 12.2 индексированная палитра цветов.

Таблица 12.2. Шестнадцатицветовая палитра VGA

№	Цвет	Значение	№	Цвет	Значение
0	Черный	#000000	8	Серый	#202020
1	Синий	#200000	9	Светло-синий	#3F0000
2	Зеленый	#002000	10	Светло-зеленый	#003F00
3	Голубой	#202000	11	Светло-голубой	#3F3F00
4	Красный	#000020	12	Светло-красный	#00003F
5	Фиолетовый	#200020	13	Светло-фиолетовый	#3F003F
6	Коричневый	#002020	14	Желтый	#003F3F
7	Белый	#303030	15	Ярко-белый	#3F3F3F

В отличие от полной RGB-палитры при задании значения цвета в режиме VGA используется только 6 наименее значащих бита каждого из RGB-байтов:

синий байт	*зеленый байт*	*красный байт*
00bbbbbb	00gggggg	00rrrrrr

Поэтому в режиме VGA можно задать только 64 * 64 * 64 = 262144 цветов (256 K). Причем одновременно может быть задано не более 256. Палитра из 256 цветов может быть задана при разрешении 320 * 200 пикселей. При разрешении 640 * 480 пикселей в режиме VGA можно одновременно задать 2 или 16 цветов. Число цветов в палитре задается компонентом *wc.numcolors* в типе *windowconfig* и последующим выполнением функции SETWINDOWCONFIG.

Соответствие между индексом (номером) цвета в палитре и его значением можно изменить, обратившись к функциям

REMAPPALETTERGB (*index, color*)

или

REMAPALLPALETTERGB (*colors*)

color - значение цвета, которому будет присвоен номер цвета палитры, заданный параметром *index*.

index - номер цвета в палитре, устанавливаемый в *color*.

colors - массив, содержащий значения цветов.

Тип параметров *color, index, colors* - INTEGER(4).

Функция REMAPPALETTERGB:

- возвращает при успешном завершении предыдущее значение цвета для заданного номера и -1 в случае неудачи;
- приписывает номер цвета значению цвета, поддерживаемому в текущем видеорежиме.

Функция REMAPALLPALETTERGB:

- возвращает 0 при успешном выполнении и -1 в случае ошибки;
- переназначает одновременно один или более доступных в установленном видеорежиме номеров цветов.

Обе функции немедленно обновляют текущие цвета объектов в соответствии с новой палитрой. Тип функций - INTEGER(4).

Функция REMAPALLPALETTERGB может изменить значения цветов не более чем для 236 индексов. Двадцать индексов резервируется для нужд системы. Если задано значение цвета, не находящегося в палитре, то при выводе графических примитивов выполняется аппроксимация значения и из палитры выбирается наиболее близкий к заданному цвет.

Изменение палитры не может быть выполнено на адаптерах с числом цветов 64 K, SVGA-адаптерах и машинах, использующих полную систему RGB-цветов.

На машинах с VGA если вы, работая в многооконном режиме QuickWin, изменили значения всех цветов палитры и использовали ее для графического вывода в одном окне, то в другом дочернем окне попытка изменения палитры приведет к изменению цвета в первом дочернем окне.

Замечание. На машине, поддерживающей более 256 цветов, нельзя выполнить анимацию путем изменения палитры. Windows 95 и Windows NT создают логическую палитру, соответствующую палитре, которая установлена аппаратно. Для палитры в 256 цветов и менее выполняется (при помощи функций REMAPPALETTERGB и REMAPALLPALETTERGB) ее непосредственное изменение. Для палитры с числом цветов более 256 изменения выполняются в неиспользованной при графическом выводе части палитры. Таким образом, текущие цвета не изменяются и, следовательно, палитровая анимация невозможна.

Для значений цветов в 16-цветовой палитре VGA в модуле MSFLIB определены именованные константы: $BLACK, $BLUE, $GREEN, $CYAN, $RED, $MAGENTA, $BROWN, $WHITE, $GRAY, $LIGHTBLUE, $LIGHTGREEN, $LIGHTCYAN, $LIGHTRED, $LIGHTMAGENTA, $YELLOW, $BRIGHTWHITE.

Все графические режимы адаптера VGA воспроизводятся на аналоговых VGA-мониторах. На монохромных мониторах цвета воспроизводятся как оттенки серого цвета.

Пример. Продемонстрировать различные оттенки синего, зеленого, красного, серого, бирюзового, фиолетового и салатового цветов.

Выведем первоначально 8 горизонтальных полос, закрасив их в цвета загружаемой по умолчанию палитры. Затем, изменяя палитру, выведем поочередно требуемую картину цветов.

В программе учтено, что серый цвет получается в результате смешения красного, зеленого и синего одинаковой интенсивности. Бирюзовый - в результате смешения цветов синего и зеленого с равной интенсивностью, и так далее.

```
use msflib
integer(2) st2, dy /20/, y /0/, k, ic
integer(4) st4, i, pal(0:15) /16*0/
logical res
type (windowconfig) wc
wc.numxpixels = 640
wc.numypixels  = 480
wc.numcolors  = 256
wc.numtextcols = -1
wc.numtextrows = -1
wc.fontsize = -1
wc.title = 'Palette test'c
res = setwindowconfig(wc)
if(.not. res) stop 'Cannot run palette test'
do k = 8, 15                    ! Цикл формирования горизонтальных полос
 st2 = setcolor(k)              ! k - номер (индекс) цвета в палитре
 st2 = rectangle($gfillinterior, 0, y, 639, y + dy)
 y = y + dy
enddo
read *                          ! Просмотр результата до нажатия Enter
do ic = 1, 7                    ! Цикл вывода оттенков разных цветов
 k = 8                          ! Изменяем значение для цвета с номером k
 do i = 140, 255, 15            ! Формирование значений цветов
  select case(ic)
```

```
  case(1)
   pal(k) = rgbtointeger(0, 0, i)  ! Синий
  case(2)
   pal(k) = rgbtointeger(0, i, 0)  ! Зеленый
  case(3)
   pal(k) = rgbtointeger(i, 0, 0)  ! Красный
  case(4)
   pal(k) = rgbtointeger(i, i, i)  ! Серый
  case(5)
   pal(k) = rgbtointeger(0, i, i)  ! Бирюзовый
  case(6)
   pal(k) = rgbtointeger(i, 0, i)  ! Фиолетовый
  case(7)
   pal(k) = rgbtointeger(i, i, 0)
  endselect
  k = k + 1
 enddo
 st4 = remapallpalettergb(pal)    ! Изменение цветовой палитры
 read *                           ! Ожидаем нажатия Enter
enddo
end
```

Вариант последовательного изменения цветов полос.

```
...
do ic = 1, 7
 k = 8
 do i = 140, 255, 15
  select case(ic)
  case(1)
   st4 = remappalettergb (k, rgbtointeger(0, 0, i))        ! Синий
  case(2)
   st4 = remappalettergb (k, rgbtointeger(0, i, 0))        ! Зеленый
  case(3)
   st4 = remappalettergb (k, rgbtointeger(i, 0, 0))        ! Красный
  case(4)
   st4 = remappalettergb (k, rgbtointeger(i, i, i))        ! Серый
  case(5)
   st4 = remappalettergb (k, rgbtointeger(0, i, i))        ! Бирюзовый
  case(6)
   st4 = remappalettergb (k, rgbtointeger(i, 0, i))        ! Фиолетовый
  case(7)
   st4 = remappalettergb (k, rgbtointeger(i, i, 0))
  endselect
  k = k + 1
 enddo
 read *
enddo
...
```

12.8.3. Не RGB-функции управления цветом

Для управления цветом в FPS существует два рода функций:

- функции, позволяющие задать любой из RGB-цветов (RGB-функции) и устанавливающие цвет по его значению;
- функции, устанавливающие цвет по его номеру (индексу) в цветовой палитре (не RGB-функции).

К первым относятся функции SETTEXTCOLORRGB, SETBKCOLORRGB и SETCOLORRGB. Ко вторым - SETTEXTCOLOR, SETBKCOLOR и SETCOLOR.

Число цветов, которые можно задать не RGB-функциями, зависит от установленной конфигурации окна, но не может в любом случае превышать 256.

12.8.3.1. Управление цветом фона

Задание текущего цвета фона по его номеру в цветовой палитре выполняется функцией

SETBKCOLOR (*index*)

index - параметр типа INTEGER(4), равный номеру цвета устанавливаемого фона.

Функция возвращает значение типа INTEGER(4), равное номеру цвета старого фона. По умолчанию устанавливается черный цвет фона.

Номер (INTEGER(4)) текущего цвета фона возвращается функцией

GETBKCOLOR ()

При выводе текста подпрограммой OUTTEXT функция SETBKCOLOR задает фон, на котором выводится текст.

Если же после вызова функции SETBKCOLOR вызвать подпрограмму CLEARSCREEN, то выполнится очистка заданной области и ее заполнение заданным цветом фона.

Пример 1. Вывести в текстовом видеорежиме строку "Проба фона", используя разные цвета текста и фона.

```
use msflib                          ! Пусть сумма номеров цвета текста и фона равняется 15
integer(2) status2
integer(4)  i, status4
character(15) st /'Background test'/
type (rccoord) rc                   ! Режим 800*600, 16 цветов
do i = 0, 15                        ! 100*37 в текстовом режиме
 status4 = setbkcolor(i)            ! Цвет фона
 status2 = settextcolor(int2(15 - i))                ! Цвет текста
 call settextposition(int2(10 + i), 40, rc)
 call outtext(st)
enddo
end
```

Пример 2. Просмотреть ярко-белый квадрат на разном фоне.

```
use msflib                          ! Режим 800*600, 16 цветов
integer(2) status2
integer(4) status4, i
status2 = setcolor(15)              ! Ярко-белый цвет
do i = 1, 14
 status4 = setbkcolor(i)            ! Цвет фона
 call clearscreen($gclearscreen)    ! Заполнение экрана цветом фона
 status2 = rectangle($gfillinterior, 350, 250, 450, 350)
 read(*, *)                         ! Ждем нажатия Enter
enddo
end
```

12.8.3.2. Управление цветом неграфического текста

Номер цвета (см. табл. 12.2), выводимого подпрограммой OUTTEXT и операторами WRITE и PRINT текста, устанавливается функцией

SETTEXTCOLOR (*index*)

index - параметр типа INTEGER(2), задающий номер (индекс) цвета.

По умолчанию устанавливается цвет с номером 15, который, если не была изменена палитра, ассоциируется с ярко-белым цветом.

Функция возвращает значение типа INTEGER(2), которое равно номеру предыдущего цвета текста.

Установленный функцией SETTEXTCOLOR цвет не воспринимается подпрограммой OUTGTEXT.

В графическом режиме диапазон номеров цветов зависит от выбранного видеорежима и может быть определен после вызова функции GETWINDOWCONFIG. Функции SETTEXTCOLOR и SETCOLOR работают в графическом режиме в одном и том же диапазоне цветов.

Номер заданного SETTEXTCOLOR цвета возвращается функцией

GETTEXTCOLOR ()

Тип возвращаемого результата - INTEGER(2).

Пример. Вывести текущую конфигурацию видеоокна в текстовом окне.

```
use msflib
logical  res
integer(2) status2
integer(4) status4
character(15) st*15
type (windowconfig) w
type (rccoord) rc                    ! Режим 800*600, 16 цветов
res = getwindowconfig (w)            ! и 100*37 для текста
status4 = setbkcolor(1)              ! Цвет фона видеоокна - синий
call clearscreen($gclearscreen)      ! Заливка экрана цветом фона
call settextwindow (15, 35, 19, 65)! Задание текстового окна
status4 = setbkcolor(3)              ! Цвет фона окна - голубой
call clearscreen($gwindow)           ! Заливка окна цветом фона
status2 = settextcolor(14)           ! Цвет текста - желтый
write(st, '(i3, a, i3)') w.numxpixels, ' * ', w.numypixels
call outtext('Resolution: ' // st )  ! 800 * 600
call settextposition(2, 1, rc)
write(st, '(i3, a, i3)') w.numtextcols, ' *', w.numtextrows
call outtext('Text screen size: ' // st)                    ! 100 * 37
call settextposition(3, 1, rc)
call outtext('Number of colors: ' // stval(w.numcolors))    ! 16
call outtext('Bit per pixel: ' // stval(w.bitsperpixel))    ! 1
contains
 character(5) function stval(val)
  integer(2) val, i / 2 /
  i = i + 1                          ! Номер ряда в текстовом окне
  call settextposition(i, 1, rc)     ! Изменяем позицию вывода
  write(stval, '(i3)') val           ! Преобразование число - строка
 end function                        ! устанавливает возвращаемое значение
end
```

12.8.3.3. Управление цветом графических примитивов

Текущий цвет выводимых графических элементов: не RGB-пикселя, отрезка прямой, прямоугольника, многоугольника, эллипса, дуги, сектора, графического текста и не RGB-заливки - может быть задан не RGB-функцией

SETCOLOR (*index*)

index - номер устанавливаемого цвета (параметр типа INTEGER(2)).

Функция при успешном выполнении возвращает значение типа INTEGER(2), равное номеру предыдущего цвета или -1 в случае неудачи. Устанавливаемый по умолчанию текущий цвет имеет максимальный номер в текущей палитре.

Номер текущего цвета можно определить, обратившись к функции

GETCOLOR ()

Тип функции - INTEGER(2).

12.8.4. RGB-функции управления цветом

12.8.4.1. Управление RGB-цветом фона

Текущий цвет фона по его значению устанавливается функцией

SETBKCOLORRGB(*color*)

color - параметр типа INTEGER(4), равный RGB-значению цвета устанавливаемого фона.

Функция возвращает величину типа INTEGER(4), равную RGB-значению цвета старого фона.

Значение (INTEGER(4)) текущего цвета фона возвращается функцией

GETBKCOLORRGB ()

При выводе текста посредством OUTTEXT функция SETBKCOLORRGB задает фон, на котором выводится текст.

Если же после вызова функции SETBKCOLORRGB вызвать подпрограмму CLEARSCREEN, то выполнится очистка заданной области и ее заполнение заданным RGB-цветом фона.

Пример. Заполнить поочередно видеоокно оттенками фиолетового цвета.

```
use msflib
integer(4) oldcolor, k
do k = 120, 255, 15                    ! Фиолетовый - наложение красного и синего
 oldcolor = setbkcolorrgb(rgbtointeger (k, 0, k))
 call clearscreen($gclearscreen)
 read *                                ! Ожидаем нажатия Enter
enddo
end
```

12.8.4.2. Управление RGB-цветом неграфического текста

Значение RGB-цвета, выводимого подпрограммой OUTTEXT и операторами WRITE и PRINT текста, устанавливается функцией

SETTEXTCOLORRGB (*color*)

color - параметр типа INTEGER(4), задающий значение цвета.

Функция возвращает величину типа INTEGER(4), которая равна значению предыдущего цвета текста.

Установленный SETTEXTCOLORRGB цвет не оказывает влияния на работу подпрограммы OUTGTEXT.

Номер заданного SETTEXTCOLORRGB цвета возвращается функцией

GETTEXTCOLORRGB ()

Тип возвращаемого результата - INTEGER(4).

12.8.4.3. Управление RGB-цветом графических примитивов

Текущий цвет выводимых графических элементов: отрезка прямой, прямоугольника, многоугольника, эллипса, дуги, сектора, графического текста - может быть задан RGB-функцией

SETCOLORRGB (*color*)

color - RGB-значение типа INTEGER(4) устанавливаемого цвета.

Функция при успешном выполнении возвращает значение типа INTEGER(4), равное значению предыдущего цвета.

Значение текущего цвета можно определить, вызвав функцию

GETCOLORRGB ()

Тип возвращаемого результата INTEGER(4).

Пример. Создать 10 вертикальных полос из различных оттенков серого цвета.

```
use msflib                          ! Режим 800*600
integer(4) :: st4, k = 105          ! k задает интенсивность серого цвета
integer(2) :: x = 0, col            ! Серый цвет - наложение оттенков красного,
do col = 1, 10                      ! зеленого и синего цветов
 st4 = setcolorrgb(rgbtointeger(k, k, k))
 st4 = rectangle($gfillinterior, x, 0, x + 80, 599)
 x = x + 80
 k = k + 15
enddo
end
```

12.9. Текущая позиция графического вывода

Графические примитивы текст и отрезок прямой выводятся начиная от текущей позиции. После перехода в графический режим текущая позиция находится в центре видеоокна. Изменение текущей позиции выполняется подпрограммами

CALL MOVETO (*x, y, t*)

или

CALL MOVETO_W (*wx, wy, wt*)

x, y - видовые координаты новой позиции типа INTEGER(2).

wx, wy - оконные координаты новой позиции типа REAL(8).

t - видовые координаты предшествующей позиции (переменная типа *xycoord*). Тип *xycoord* определен в модуле MSFLIB:

```
type xycoord
 integer(2) xcoord          ! x-координата
 integer(2) ycoord          ! y-координата
end type xycoord
```

wt - оконные координаты предшествующей позиции (переменная типа w*xycoord*). Тип w*xycoord* определен в модуле MSFLIB:

```
type wxycoord
 real(8) wx                 ! x-оконная координата
 real(8) wy                 ! y-оконная координата
end type wxycoord
```

При изменении текущей позиции изображение не меняется. Подпрограмма MOVETO изменяет текущую позицию в видовой, а MOVETO_W - в оконной системе координат.

Замечания:

1. В графическом режиме могут быть определены две позиции вывода: позиция вывода не зависящего от шрифта текста (выводится подпрограммой OUTTEXT), устанавливаемая в текстовом окне функцией SETTEXTPOSITION, и текущая позиция графического вывода, задаваемая в видовой или оконной системе координат.

2. Текущая позиция графического вывода меняется после вывода двух графических примитивов: отрезка прямой и текста.

Координаты текущей позиции можно получить, обратившись к подпрограммам

GETCURRENTPOSITION (*t*)

или

GETCURRENTPOSITION_W (*wt*)

t и *wt* - переменные производного типа *xycoord* и w*xycoord*, содержащие соответственно видовые и оконные координаты текущей позиции. Описание типов *xycoord* и w*xycoord* приведено выше.

Когда создается окно, текущая графическая позиция находится в его центре. Затем текущая позиция графического вывода может быть изменена графическими процедурами LINETO, MOVETO и OUTGTEXT.

12.10. Графические примитивы

Большинство графических объектов может быть изображено при помощи небольшого числа простых графических элементов, называемых *графическими примитивами*.

Процедуры графической библиотеки FPS, позволяющие рисовать графические примитивы, приведены в табл. 12.3. Графические элементы изображаются в заданной программистом системе координат и в заданном окне вывода.

Таблица 12.3. Процедуры вывода графических элементов

Процедуры	*Отображаемые графические элементы*
SETPIXEL, SETPIXEL_W	Пиксель (закраска пикселя)
SETPIXELS (подпрограмма)	Группа пикселей
SETPIXELRGB, SETPIXELRGB_W	Пиксель (закраска пикселя RGB-цветом)
SETPIXELSRGB (подпрограмма)	Группа RGB-пикселей
LINETO, LINETO_W	Отрезок прямой
RECTANGLE, RECTANGLE_W	Прямоугольник
POLYGON, POLYGON_W	Многоугольник
ELLIPSE, ELLIPSE_W	Эллипс или окружность
ARC, ARC_W	Дуга окружности
PIE, PIE_W	Сектор окружности
FLOODFILLRGB, FLOODFILRGB_W, FLOODFILL, FLOODFIL_W	Заполнение (заливка) замкнутой области заданным цветом и по заданному шаблону
OUTGTEXT (подпрограмма)	Зависимый от шрифта (графический) текст

Для графического примитива могут быть установлены следующие характеристики (атрибуты):

- тип линии (например, сплошная, пунктирная, штрихпунктирная);
- цвет линии;
- способ вывода (для отрезков, прямоугольников и многоугольников);
- шаблон и цвет заполнения (для замкнутых геометрических фигур).

Кроме того, вывод примитива может выполняться на заранее установленном фоне, то есть после заливки окна вывода заданным цветом. Цвет фона задается функциями SETBKCOLORRGB и SETBKCOLOR. Цвет графических примитивов - функциями SETCOLORRGB и SETCOLOR. Закраска видеоокна, видового порта или окна вывода цветом фона выполняется подпрограммой CLEARSCREEN. Более подробно вопросы управления цветом рассмотрены в разд. 12.8.

По умолчанию цвет фона - черный, цвет линий - белый.

Функции, оканчивающиеся на _W, применимы только в оконной системе координат. И наоборот, функции, не имеющие этого окончания, используются в видовой или физической системе координат.

Текущий цвет графических примитивов (кроме примитивов, выводимых SETPIXEL, SETPIXELS, SETPIXELRGB, SETPIXELSRGB) устанавливается функциями SETCOLORGB и SETCOLOR.

12.10.1. Вывод пикселей

Для закраски пикселей существует два рода процедур:

- не RGB-процедуры, закрашивающие пиксель цветом, задаваемым по его номеру в цветовой палитре (разд. 12.8.2);

- RGB-процедуры, закрашивающие пиксель любым RGB-цветом.

Не RGB-функции закраски пикселя текущим цветом:

SETPIXEL (*x, y*)

или

SETPIXEL_W (*wx, wy*)

x, y - видовые координаты пикселя - INTEGER(2).

wx, wy - оконные координаты пикселя - REAL(8).

Функция возвращает значение типа INTEGER(2), которое в случае успеха равно номеру цвета, в который был окрашен пиксель до выполнения функции, или -1 в случае ошибки (при задании координат за пределами окна вывода). Номер цвета вывода пикселя задается функцией SETCOLOR.

Номер цвета пикселя можно узнать, обратившись к функциям

GETPIXEL (*x, y*)

или

GETPIXEL_W (*wx, wy*)

Смысл и тип параметров *x, y* и *wx, wy* такой же, какой имеют и одноименные параметры функций SETPIXEL и SETPIXEL_W.

Функция возвращает значение типа INTEGER(2), которое равно номеру цвета пикселя в случае успеха или -1 в противном случае. Неудача может произойти, если указанные координаты находятся за пределами окна вывода. Соответствие между номером цвета и его значением можно изменить функциями REMAPPALETTERGB или REMAPALLPALETTERGB.

Группу пикселей можно отобразить, вызвав не RGB-подпрограмму

CALL SETPIXELS (*n, x, y, color*)

n - параметр типа INTEGER(4), задающий число отображаемых пикселей и число элементов в массивах *x, y* и *color*.

x, y - массивы типа INTEGER(2), содержащие *x* и *y* координаты, отображаемых пикселей. Элементы *x*(1) и *y*(1) содержат координаты первого пикселя, *x*(2) и *y*(2) - второго и так далее.

color - массив типа INTEGER(2), содержащий номера цветов (разд. 12.8.2) отображаемых пикселей.

Номера цветов выведенной группы пикселей возвращаются параметром *color* подпрограммы

CALL GETPIXELS (*n, x, y, color*)

Параметры *n, x, y* имеют тот же смысл, что и в подпрограмме SETPIXELS.

color - массив типа INTEGER(2), в который записываются номера цветов пикселей, заданных массивами *x* и *y*.

Пример.

```
use msflib                          ! Режим 800*600, 16 цветов
integer, parameter :: n = 1000
integer(2) x(n), y(n), color(n), i, sv / 1 /, k / 1 /
real ran
do i = 1, n
 sv = -sv                           ! Обеспечим чередование знака
 call random(ran)
 x(i) = 400 + sv * int2(100 * ran)
 call random(ran)
 y(i) = 300 + sv * int2(100 * ran)
 color(i) = k
 k = k + 1
 if (k == 16) k = 1
end do
call setpixels(n, x, y, color)      ! Закраска n пикселей
end
```

Не RGB-процедуры обеспечивают доступ к загружаемой цветовой палитре, которая в лучшем случае может содержать 256 цветов (число цветов в палитре можно определить, вызвав функцию GETWINDOWCONFIG). RGB-процедуры вывода пикселя позволяют вывести пиксель любого возможного на данном графическом адаптере RGB-цвета.

RGB-функции закраски пикселя:

SETPIXELRGB (*x*, *y*, *color*)

или

SETPIXELRGB_W (*wx*, *wy*, *color*)

x, *y* - видовые координаты типа INTEGER(2) выводимого пикселя.

wx, *wy* - оконные координаты типа REAL(8) выводимого пикселя.

color - значение типа INTEGER(4) RGB-цвета (разд. 12.8.1), в который закрашивается пиксель.

Функция возвращает значение типа INTEGER(4), которое равно предыдущему значению RGB-цвета.

Если пиксель находится за пределами области вывода, то он игнорируется.

Номер цвета пикселя можно узнать, обратившись к функциям

GETPIXELRGB (*x*, *y*)

или

GETPIXELRGB_W (*wx*, *wy*)

Смысл и тип параметров *x*, *y* и *wx*, *wy* такой же, что и у одноименных параметров функций SETPIXELRGB и SETPIXELRGB_W.

Функция возвращает значение типа INTEGER(4), которое равно RGB-значению цвета пикселя в случае успеха или -1 в противном случае (например, если координаты находятся за пределами окна вывода).

Пример.

```
use msflib                          ! Режим 800*600, 16 цветов
integer(2) x, y / 100 /
integer(4) color, oldcolor, i
do i = 1, 3
 select case (i)
 case(1)
   color = #0000ff                  ! Красный цвет
 case(2)
   color = #00ff00                  ! Зеленый цвет
 case (3)
   color = #ff0000                  ! Синий цвет
 end select
 y = y + 100
 do x = 100, 700, 2                 ! Вывод пикселей
   oldcolor = setpixelrgb(x, y, color)
 enddo
enddo
end
```

Массив RGB-пикселей выводится подпрограммой

CALL SETPIXELSRGB (*n*, *x*, *y*, *color*)

Параметры *n*, *x*, *y* имеют тот же смысл, что и в подпрограмме SETPIXELS.

color - массив типа INTEGER(4), содержащий RGB-значения цветов (разд. 12.8.1) отображаемых пикселей.

Номера цветов выведенной группы пикселей возвращаются параметром *color* подпрограммы

CALL GETPIXELSRGB (*n*, *x*, *y*, *color*)

Параметры *n*, *x*, *y* имеют тот же смысл, что и в подпрограмме SETPIXELS.

color - массив типа INTEGER(4), в который записываются значения RGB-цветов пикселей, заданных массивами *x* и *y*.

12.10.2. Вывод отрезка прямой линии

Функции вывода отрезка прямой:

LINETO (*x*, *y*)

или

LINETO_W (*wx*, *wy*)

x, *y* - видовые координаты конечной точки отрезка - INTEGER(2).

wx, *wy* - оконные координаты конечной точки отрезка - REAL(8).

Функция возвращает значение типа INTEGER(2), которое, если отрезок нарисован, отлично от нуля или равно нулю в противном случае.

Отрезок выводится от текущей позиции до конечной точки, задаваемой параметрами *x*, *y* или *wx*, *wy*. При выводе отрезка используется текущий цвет, задаваемый функциями SETCOLORRGB или SETCOLOR, текущий способ вывода линий, задаваемый функцией SETWRITEMODE, и

текущий тип линии, задаваемый подпрограммой SETLINESTYLE (разд. 12.12).

Способ вывода влияет на получаемый при выводе цвет отрезка (разд. 12.14). По умолчанию отрезки отображаются текущим цветом.

Отрезки можно выводить, применяя разные маски вывода, которые задают тип линии, например штриховая линия, штрихпунктирная, пунктирная и так далее. По умолчанию выводятся сплошные линии.

В случае успешного выполнения LINETO или LINETO_W текущей становится соответствующая конечной точке отрезка позиция (*x, y* или *wx, wy*).

Замечание. Если область, границы которой образованы выведенными функцией LINETO отрезками прямых, заполняется посредством FLOODFILLRGB или FLOODFILL (разд. 12.13), то для отображения границ области необходимо использовать сплошной тип линии.

12.10.3. Вывод прямоугольника

Вывод прямоугольника выполняется функциями

RECTANGLE (*control*, *x*1, *y*1, *x*2, *y*2)

или

RECTANGLE_W (*control*, *wx*1, *wy*1, *wx*2, *wy*2)

control - флаг заполнения прямоугольника - INTEGER(2).

*x*1, *y*1 - видовые координаты левого верхнего угла прямоугольника.

*x*2, *y*2 - видовые координаты правого нижнего угла прямоугольника. Тип параметров *x*1, *y*1, *x*2, *y*2 - INTEGER(2).

*wx*1, *wy*1 - оконные координаты левого верхнего угла прямоугольника.

wx2, wy2 - оконные координаты правого нижнего угла прямоугольника. Тип параметров *wx*1, *wy*1, *wx*2, *wy*2 - INTEGER(2).

Функция возвращает значение типа INTEGER(2), которое отлично от нуля в случае успеха или равно нулю в случае неудачи.

Границы прямоугольника рисуются с использованием текущего цвета, текущего способа вывода и текущего типа линии (разд. 12.12).

Параметр *control* может принимать значения определенных в модуле MSFLIB именованных констант $GFILLINTERIOR и $GBORDER.

Если *control* равен $GFILLINTERIOR, то осуществляется заполнение прямоугольника с использованием текущего цвета и шаблона (маски) заполнения, устанавливаемого подпрограммой SETFILLMASK (разд. 12.13). По умолчанию осуществляется сплошное заполнение замкнутой области.

Если в *control* установлено значение $GBORDER, то выводятся только границы прямоугольника с использованием текущих цвета, способа вывода и типа линии.

Замечание. Заполнение прямоугольника цветом, отличным от цвета границы, может быть выполнено посредством функций FLOODFILLRGB

или FLOODFILL, причем такое заполнение возможно, если прямоугольник выведен с использованием сплошного типа линии.

12.10.4. Вывод многоугольника

Произвольной формы многоугольник выводится функциями

POLYGON (*control*, *ppoints*, *cpoints*)

или

POLYGON_W (*control*, *wppoints*, *cpoints*)

Параметр *control* принимает те же значения и имеет тот же смысл и тип, что и одноименный параметр функций RECTANGLE.

ppoints - массив типа *xycoord*, состоящий из элементов, задающих ломаную линию.

cpoints - параметр типа INTEGER(2), задающий число точек ломаной линии.

wppoints - массив типа *wxycoord* из элементов, задающих ломаную линию.

Типы *xycoord* и *wxycoord* определены в модуле MSFLIB. Их описание дано в разд. 12.6.

Функция возвращает значение типа INTEGER(2), которое отлично от нуля при успешном выполнении или равно нулю в противном случае.

Границы многоугольника вычерчиваются с использованием текущего цвета, текущего способа вывода и текущего типа линии (разд. 12.8 и 12.12).

Параметр *cpoints* указывает число используемых точек массива *ppoints* или *wppoints*. Последняя выводимая точка массива всегда соединяется отрезком прямой с первой точкой.

Замечание. Заполнение многоугольника цветом, отличным от цвета границы, может быть выполнено посредством функций FLOODFILLRGB или FLOODFILL, причем такое заполнение возможно, если многоугольник выведен с использованием сплошного типа линии.

Пример. Изобразить песочные часы, заполнив их пылинками песка.

```
use msflib                          ! Режим 800*600, 16 цветов
integer(1) :: smask(8) = (/ 128, 4, 32, 2, 16, 8, 0, 64 /)
integer(2) status2
type(xycoord) sclock(6)
sclock.xcoord = (/ 300, 500, 410, 500, 300, 390 /)
sclock.ycoord = (/  10,  10, 300, 590, 590, 300 /)
status2 = setcolor(14)
call setfillmask(smask)             ! Задаем шаблон заполнения
status2 = polygon($gfillinterior, sclock, 6)
end
```

Пояснение. Для имитации заполнения часов пылинками песка создаем шаблон (маску) заполнения (массив *smask*). Текущая маска заполнения устанавливается подпрограммой SETFILLMASK (разд. 12.13).

12.10.5. Вывод эллипса и окружности

Изображение эллипса и окружности выполняется функциями

ELLIPSE (*control*, *x*1, *y*1, *x*2, *y*2)

или

ELLIPSE_W (*control*, *wx*1, *wy*1, *wx*2, *wy*2)

control - флаг типа INTEGER(2) заполнения эллипса.

*x*1, *y*1 - видовые координаты типа INTEGER(2) левого верхнего угла ограничивающего эллипс прямоугольника.

*x*2, *y*2 - видовые координаты типа INTEGER(2) правого нижнего угла ограничивающего эллипс прямоугольника.

*wx*1, *wy*1 - оконные координаты типа REAL(8) левого верхнего угла ограничивающего эллипс прямоугольника.

wx2, wy2 - оконные координаты типа REAL(8) правого нижнего угла ограничивающего эллипс прямоугольника.

Функция возвращает значение типа INTEGER(2), которое отлично от нуля при успешном выполнении или равно нулю в противном случае.

Граница эллипса вычерчивается с использованием текущего цвета. Такие атрибуты, как способ вывода и тип линии, для эллипса не устанавливаются. Вывод эллипса всегда выполняется с применением сплошного типа линии.

Параметр *control* принимает те же значения и имеет тот же смысл, что и одноименный параметр функций RECTANGLE и POLYGON.

Замечание. Заполнение эллипса цветом, отличным от цвета границы, может быть выполнено посредством функций FLOODFILLRGB или FLOODFILL.

12.10.6. Вывод дуги эллипса и окружности

Дуга эллипса или окружности выводится функциями

ARC (*x*1, *y*1, *x*2, *y*2, *x*3, *y*3, *x*4, *y*4)

или

ARC_W (*wx*1, *wy*1, *wx*2, *wy*2, *wx*3, *wy*3, *wx*4, *wy*4)

*x*1, *y*1 - видовые координаты типа INTEGER(2) левого верхнего угла ограничивающего прямоугольника.

*x*2, *y*2 - видовые координаты типа INTEGER(2) правого нижнего угла ограничивающего прямоугольника.

*x*3, *y*3 - начальный вектор - INTEGER(2).

*x*4, *y*4 - конечный вектор - INTEGER(2).

*wx*1, *wy*1 - оконные координаты типа REAL(8) левого верхнего угла ограничивающего прямоугольника.

*wx*2, *wy*2 - оконные координаты типа REAL(8) правого нижнего угла ограничивающего прямоугольника.

*wx*3, *wy*3 - начальный вектор - REAL(8).

*wx*4, *wy*4 - конечный вектор - REAL(8).

Функция возвращает значение типа INTEGER(2), которое отлично от нуля при успешном выполнении или равно нулю в случае неудачи.

Центр дуги находится в центре ограничивающего прямоугольника (рис. 12. 4).

Начальный вектор - это вектор, проходящий, через центр дуги и точку *x*3, *y*3 (*wx*3, *wy*3) (рис. 12. 4).

Конечный вектор - это вектор, проходящий через центр дуги и точку *x*4, *y*4 (*wx*4, *wy*4) (рис. 12. 4).

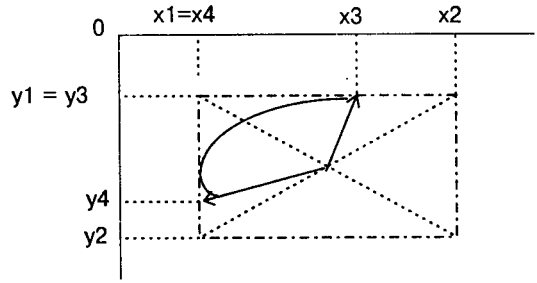

Рис.12.4. Параметры функции ARC в видовой системе координат

Дуга вычерчивается с использованием текущего цвета. Такие атрибуты, как способ вывода и тип линии, для дуги не устанавливаются.

Дуга рисуется как часть эллипса от точки пересечения эллипса с начальным вектором до точки пересечения эллипса с конечным вектором. Координаты точек пересечения эллипса с начальным и конечным векторами возвращаются функцией GETARCINFO. Вывод дуги выполняется против часовой стрелки.

12.10.7. Вывод сектора

Сектор выводится функциями

PIE (*control*, *x*1, *y*1, *x*2, *y*2, *x*3, *y*3, *x*4, *y*4)

или

PIE_W (*control*, wx1, wy1, wx2, wy2, wx3, wy3, wx4, wy4)

Параметр *control* принимает те же значения и имеет тот же смысл и тип, что и одноименный параметр функций RECTANGLE, POLYGON и ELLIPSE.

Смысл и тип параметров *x*1, *y*1, *x*2, *y*2, *x*3, *y*3, *x*4, *y*4, *wx*1, *wy*1, *wx*2, *wy*2, *wx*3, *wy*3, *wx*4, *wy*4 такой же, что и у одноименных параметров функции ARC.

Функция возвращает значение типа INTEGER(2), которое отлично от нуля при успешном выполнении или равно нулю в случае неудачи.

В отличие от дуги конечные точки сектора соединены с его центром. Центр сектора, как и центр дуги, совпадает с центром ограничивающего прямоугольника. Таким образом, сектор является замкнутой областью и,

следовательно, может быть заполнен текущим цветом с использованием текущей маски заполнения.

Граница сектора выводится против часовой стрелки с использованием текущего цвета. Так же как и для дуги, способ вывода и тип линии для сектора не устанавливаются.

Замечание. Заполнение сектора цветом, отличным от цвета границы, может быть выполнено посредством функций FLOODFILLRGB или FLOODFILL.

12.10.8. Координаты конечных точек дуги и сектора

Координаты конечных точек последней выведенной дуги или сектора можно определить, обратившись к функции

GETARCINFO (*pstart*, *pend*, *ppaint*)

pstart - координаты начальной точки дуги - тип *xycoord*.
pend - координаты конечной точки дуги - тип *xycoord*.
ppaint - координаты точки начала заливки - тип *xycoord*.
Тип *xycoord* определен в модуле MSFLIB и приведен в разд. 12.6.

Функция возвращает значение типа INTEGER(2), которое отлично от нуля, если вывод дуги или сектора был выполнен успешно (с момента последней очистки видеоокна или выбора видового порта, или установки нового графического видеорежима), или которое равно нулю в случае неудачи.

При успешном завершении функция GETARCINFO заносит в параметры *pstart* и *pend* (переменные типа *xycoord*) видовые координаты начальной и конечной точки дуги.

Дополнительно в параметр *ppaint* типа *xycoord* устанавливаются координаты, которые можно использовать для задания начальной точки заполнения сектора. Эта информация полезна, если необходимо произвести заполнение сектора цветом, который отличается от цвета границ. Для этого следует использовать функцию FLOODFILLRGB. Последовательность вывода сектора может быть такой: вывести незаполненный сектор; вызвать функцию GETARCINFO; изменить текущий цвет, применив функцию SETCOLOR; вызвать функцию FLOODFILLRGB, использовав установленные в *ppaint* координаты в качестве координат начальной точки заполнения сектора.

12.10.9. Пример вывода графических примитивов

Задать в видеоокне 4 окна вывода (рис. 12.5) и вывести: в первом окне - дугу, во втором сектор и выполнить затем его заливку, в третьем - эллипс, в четвертом заполненный прямоугольник.

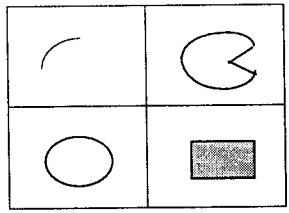

Рис. 12.5. Расположение геометрических фигур в видеоокне

```
use msflib                          ! Режим 800*600, 16 цветов
integer(2) s2, XE/800/, YE/600/
type(xycoord) ps, pe, pp
! Задание первого видового порта и заливка окна вывода цветом фона
call preps(int2(0), int2(0), int2(XE/2), int2(YE/2), 1, 14)
! Вывод дуги
s2 = arc(XE/8, YE/8, 3*XE/8, 3*YE/8, XE/4, YE/8, XE/8, YE/4)
! Задание второго видового порта и заливка окна вывода цветом фона
call preps(int2(XE/2), int2(0), int2(XE), int2(YE/2), 7, 11)
! Вывод сектора
s2 = pie($gborder,XE/8,YE/8,3*XE/8,3*YE/8,3*XE/8,YE/8,3*XE/8,3*YE/8)
s2 = getarcinfo(ps, pe, pp)         ! pp - начальная точка заливки
s2 = setcolor(8)                    ! Цвет заливки темно-серый
s2=floodfill(pp.xcoord, pp.ycoord,11)          ! Заливка сектора
! Задание третьего видового порта и заливка окна вывода цветом фона
call preps(int2(0), int2(YE/2), int2(XE/2), int2(YE), 7, 11)
! Вывод эллипса
s2 = ellipse($gborder, XE/8, YE/8, 3*XE/8, 3*YE/8)
! Задание четвертого видового порта и заливка окна вывода цветом фона
call preps(int2(XE/2), int2(YE/2), int2(XE), int2(YE), 1, 14)
! Вывод закрашенного прямоугольника
s2 = rectangle($gfillinterior, XE/8, YE/8, 3*XE/8, 3*YE/8)
contains
 subroutine preps(x1, y1, x2, y2, c1, c2)
  integer(2) x1, y1, x2, y2, status2
  integer(4) c1, c2
  call setviewport (x1, y1, x2, y2)  ! Окно вывода
  status2 = setcolor(c1)             ! Цвет заливки
  status2 = floodfill(2, 2, 1)       ! Заливка окна
  status2 = setcolor(c2)             ! Цвет объекта
 end subroutine
end
```

Пояснения:

1. Заливка выполняется методом заполнения замкнутой области с затравкой. Для заливки внутри области указывается произвольная точка и цвет границы области. Если затравочная точка указана вне замкнутой области, то будут залиты все пиксели, находящиеся за пределами области. Метод применим также и для заливки областей с отверстиями. Главное, чтобы границы области и отверстий имели один и тот же цвет.

2. Поскольку первоначально окно не содержит графических объектов, то для его заливки можно указать любой цвет границы области (третий параметр функции FLOODFILL).

12.11. Вывод текста

В графическом режиме можно выводить два вида текста: с использованием и без использования шрифтов. Последний вид текста может быть выведен также и в текстовом режиме.

12.11.1. Вывод текста без использования шрифтов

Выводимый без использования шрифта текст имеет постоянную для заданного видеорежима высоту и ширину, определяемую размером знакоместа. Каждый символ выводится в некоторую позицию видеоокна (знакоместо). Число таких позиций зависит от видеорежима и может быть определено в результате вызова функции GETWINDOWCONFIG. Результаты устанавливаются в компоненты переменной, например *t* типа *windowconfig*: *t.numtextcols* и *t.numtextrows* (разд. 12.5).

Единицами измерения координаты позиции вывода текста являются ряд (строка) и столбец видеоокна (текстового окна), на пересечении которых позиция расположена. Позиция, расположенная в левом верхнем углу видеоокна или текущего окна, имеет координату (1, 1).

В любом видеорежиме текст может быть выведен подпрограммой

CALL OUTTEXT (*text*)

text - строка выводимого текста. Строка выводится вместе с хвостовыми пробелами.

Вывод текста начинается от текущей позиции. После вывода строки текущей становится первая следующая за строкой позиция.

Подпрограмма OUTTEXT выводит только строки символов, поэтому вывод числовых или логических данных подпрограммой возможен только после записи в строку их символьного представления.

Вывод текста посредством OUTTEXT, WRITE и PRINT может быть ограничен окном, задание которого выполняется подпрограммой

CALL SETTEXTWINDOW (*r*1, *c*1, *r*2, *c*2)

*r*1, *c*1, - параметры типа INTEGER(2), определяющие левый верхний ряд и столбец окна текста.

*r*2, *c*2 - параметры типа INTEGER(2), определяющие правый нижний ряд и столбец окна текста.

Координаты окна *r*1, *c*1, *r*2, *c*2 задаются относительно верхнего левого угла видеоокна. После задания текстового окна отсчет текущей позиции выполняется уже относительно его верхнего левого угла. Текст в окне выводится начиная с верхнего левого угла окна. Если окно заполнено полностью, то последующие строки пишутся на месте последней строки.

Замечание. Подпрограмма SETTEXTWINDOW не оказывает влияния на работу подпрограммы OUTGTEXT, осуществляющей вывод графического зависимого от шрифта текста.

В заданном текстовом окне текущая позиция начала вывода текста посредством OUTTEXT, WRITE и PRINT изменяется подпрограммой

CALL SETTEXTPOSITION (*row, column, t*)

row, column - параметры типа INTEGER(2), задающие новую позицию вывода текста. Позиция вывода текста задается относительно левого верхнего угла окна.

t - переменная типа *rccoord*, содержащая координаты прежней позиции вывода текста. Тип *rccoord* определен в модуле MSFLIB:

```
type rccoord
 integer(2) row                ! координаты ряда позиции
 integer(2) col                ! координаты столбца позиции
end type rccoord
```

Текстовая позиция с координатами (1, 1) определяет левый верхний угол текстового окна. Вывод текста посредством OUTTEXT, WRITE и PRINT начинается от текущей текстовой позиции. Позиция вывода зависящего от шрифта текста не зависит от текущей текстовой позиции, а определяется текущей позицией графического вывода, возвращаемой подпрограммой GETCURRENTPOSITION.

Замечание. При совместном использовании подпрограмм SETTEXTPOSITION, OUTTEXT и оператора WRITE (PRINT) следует подавлять вывод оператором WRITE (PRINT) символов конца строки и новой строки (CHAR(13) и CHAR(10)), используя для этого преобразования \ или $ или опцию ADVANCE = 'NO'.

Текущая позиция вывода текста возвращается подпрограммой

CALL GETTEXTPOSITION (*t*)

t - переменная типа rccoord.

Текстовое окно можно заполнить фоном. Цвет фона задается функциями SETBKCOLORRGB и SETBKCOLOR. Закраска видеоокна или окна вывода цветом фона выполняется подпрограммой CLEARSCREEN (разд. 12.7). Цвет выводимого посредством OUTTEXT, WRITE и PRINT текста устанавливается функциями SETTEXTCOLORRGB и SETTEXTCOLOR.

В ранее определенном текстовом окне можно организовать прокрутку (перемещение вверх или вниз) выведенного текста, применив подпрограмму

CALL SCROLLTEXTWINDOW (*rows*)

rows - параметр типа INTEGER(2), задающий число строк прокрутки текста, то есть число, определяющее, на сколько строк видеоокна переместится вверх или вниз заданный текст. Если *rows* > 0, то организовывается прокрутка вверх. Если *rows* < 0 - вниз.

Если текстовое окно не задано, то подпрограмма организовывает прокрутку текста на всем видеоокне. Перемещенные в результате прокрутки за пределы окна данные теряются.

По умолчанию выводимый OUTTEXT текст, если его длина превышает ширину окна, переносится на следующую строку. Однако обратившись к функции

WRAPON (*option*)

можно, задав параметр *option,* равный $GWRAPOFF, установить режим, при котором не уместившиеся в окне символы отсекаются. Можно затем вернуться к режиму переноса символов. Для этого нужно вызвать WRAPON с *option*, равным $GWRAPON. Тип *option* - INTEGER(2).

Функция возвращает значение типа INTEGER(2), которое равно предыдущему значению *option*.

WRAPON не оказывает влияния на работу подпрограммы OUTGTEXT.

12.11.2. Вывод зависимого от шрифта текста

Вывод текста с использованием шрифтов выполняется в графическом видеорежиме. Использование шрифтов позволяет выводить текст разной высоты, под различными углами наклона литер и самого текста.

Для вывода использующего шрифты текста следует провести подготовительные операции: выполнить инициализацию шрифтов и установить текущий шрифт.

Инициализация шрифтов выполняется функцией

INITIALIZEFONTS ()

Функция возвращает значение типа INTEGER(2), равное числу инициализированных шрифтов.

Шрифты должны быть инициализированы до выполнения использующих шрифты функций (GETFONTINFO, GETGTEXTEXTENT, SETFONT, OUTGTEXT). Установленный шрифт оказывает влияние только на вывод, выполняемый подпрограммой OUTGTEXT, и не влияет на форму вывода операторов WRITE, PRINT и подпрограммы OUTTEXT.

Инициализация шрифтов должна выполняться для каждого окна до вызова функции SETFONT.

Установка текущего шрифта вывода текста и характеристик шрифта выполняется функцией

SETFONT (*options*)

options - параметр символьного типа, задающий имя шрифта и его характеристики.

Функция возвращает номер шрифта при успешном выполнении либо -1 в случае ошибки.

Функция загружает и делает текущим указанный параметром *options* шрифт с заданными характеристиками.

Параметр *options* содержит приведенные ниже коды характеристик шрифта. Позиция задания кодов и регистр не имеют значения.

t'*fontname*' - тип шрифта, например, "t'*courier*'", "t'*helv*'", "t'*script*'", "t:'*tms rmn*'", "t'*modern*'", "t'*roman*'"...

h*y* - высота символа в пикселях (*y* - число пикселей).

w*x* - ширина символа в пикселях (*x* - число пикселей).

f - выбираются только фиксированные шрифты (каждый символ имеет одинаковую ширину).

p - выбираются только пропорциональные шрифты (ширина символа зависит от его насыщенности, например символы *j* и *a* имеют в случае пропорционального шрифта разную ширину).

Опции *f* и *p* не могут быть использованы одновременно.

v - выбор векторного шрифта, например Roman, Modern, Script (шрифт задается как множество отрезков прямых).

r - выбор растрового (битового) шрифта, например Courier, Helvetica, Palatino (шрифт задается как множество пикселей).

Опции *v* и *r* не могут быть использованы одновременно.

e - вывод текста в формате жирного шрифта. Этот параметр игнорируется, когда вывод в формате жирного шрифта невозможен.

u - вывод подчеркнутого текста. Параметр игнорируется, если шрифт не допускает подчеркивания.

i - вывод курсива (наклонного текста). Параметр игнорируется, если шрифт не поддерживает курсива.

b - выбор шрифта, наиболее полно удовлетворяющего заданным параметрам.

n*x* - выбор шрифта, для которого число *x* меньше или равно возвращаемого функцией INTIALIZEFONTS значения. Эта опция не используется совместно с другими вышеприведенными опциями.

Число задаваемых опций произвольно (кроме случая использования опции n*x*, которая применяется отдельно). Если заданы взаимно исключающие опции, например *f* и *p* или *r* и *v*, то они игнорируются.

Если опция *b* задана и по крайней мере один шрифт инициализирован, то SETFONT всегда устанавливает шрифт и возвращает 0, что означает удачное завершение.

Приоритеты опций при выборе наиболее подходящего шрифта таковы (приведены в порядке убывания):

- высота символа (опция h*y*);
- тип шрифта;
- ширина символа (опция w*x*);
- фиксированный или пропорциональный шрифт (опции *f* и *p*).

Если заданы несуществующие высота или ширина шрифта и указана опция *b*, то функция SETFONT выберет ближайший подходящий шрифт. При этом меньшая высота имеет приоритет над большей.

Если заданы несуществующие высота или ширина, то в случае векторного шрифта (задана опция *v* или задан тип векторного шрифта) SETFONT выполнит масштабирование шрифта. Битовые шрифты не масштабируются.

Если задана опция n*x*, то SETFONT проигнорирует все иные параметры и установит шрифт, номер которого соответствует *x*.

Если задана высота шрифта и отсутствует его ширина (или наоборот), то SETFONT вычислит пропущенное значение, исходя из пропорций между шириной и высотой.

Характеристики текущего шрифта возвращаются функцией

GETFONTINFO (*font*)

font - переменная типа fontinfo. Тип задан в модуле MSFLIB:

```
TYPE fontinfo
 INTEGER(4) type              ! 1 = TrueType, 0 = Bit Map
 INTEGER(4) ascent            ! Расстояние в пикселях от базовой линии
                              ! до верхушки символов
 INTEGER(4) pixwidth          ! Ширина символа в пикселях
                              ! (0 для пропорционального шрифта)
 INTEGER(4) pixheight         ! Высота символа в пикселях
 INTEGER(4) avgwidth          ! Средняя ширина символа в пикселях
 CHARACTER(32)xfacename       ! Имя шрифта
 LOGICAL(1) italic            ! .TRUE., если текущий шрифт - курсив
 LOGICAL(1) emphasized        ! .TRUE., если текущий шрифт выводится
                              ! в жирном формате
 LOGICAL(1) underline         ! .TRUE., если текст выводится подчеркнутым
END TYPE fontinfo
```

Функция возвращает значение типа INTEGER(2), равное нулю, в случае успеха и -1 при неудаче.

После инициализации и установки шрифта можно вывести строку текста, применив подпрограмму

CALL OUTGTEXT (*text*)

text - строка выводимого текста. Строка выводится вместе с хвостовыми пробелами.

Вывод текста начинается от текущей позиции графического вывода. Текст выводится с использованием текущего шрифта и цвета. После вывода строки текущей становится первая следующая за строкой позиция. Изменение текущей позиции вывода графического текста выполняется подпрограммой MOVETO. Также текущая позиция графического вывода меняется после работы функции LINETO. Текущая позиция графического вывода возвращается подпрограммой GETCURRENTPOSITION.

При работе с подпрограммой OUTGTEXT текущий цвет текста устанавливается функциями SETCOLORRGB и SETCOLOR.

Длина выводимой посредством OUTGTEXT строки текста в пикселях (включая и хвостовые пробелы) возвращается функцией

GETGTEXTEXTENT (*text*)

text - символьная переменная, содержащая строку текста, длина которой должна быть определена.

Функция возвращает длину текста *text* в пикселях. В случае неудачи (если шрифты не были установлены посредством INITIALIZEFONTS) функция возвращает -1. Тип возвращаемого значения - INTEGER(2).

Перед обращением к GETGTEXTEXTENT шрифты должны быть инициализированы. Длина текста вычисляется с учетом выбранного шрифта.

Ориентация выводимого подпрограммой OUTGTEXT текста может быть изменена в результате вызова подпрограммы

CALL SETGTEXTROTATION (*degrees*)

degrees - параметр типа INTEGER(4), задающий угол поворота текста (его ориентацию) в десятках градусов (число градусов, умноженное на 10). Отсчет угла поворота текста выполняется против часовой стрелки.

По умолчанию текст выводится в горизонтальной ориентации (*degrees* равен нулю). Если параметр *degrees* равен 900, то выводится текст, повернутый против часовой стрелки на 90° текста. Задание 1800 (180°) означает вывод справа налево перевернутого текста и так далее. При задании значения большего 3600 (360°) подпрограмма берет значение

MODULO (Заданное_значение_*degrees*, 3600)

Минимальное изменение ориентации текста - 1°.

Текущее значение ориентации текста возвращается функцией

GETGTEXTROTATION ()

Функция возвращает число типа INTEGER(4), равное установленной подпрограммой SETGTEXTROTATION ориентации текста в десятках градусов (число градусов, умноженное на 10).

Пример. Вывести имена шрифтов в различных ориентациях.

```
use msflib                          ! Режим 800*600, 16 цветов
integer, parameter :: nfonts = 6
integer(2) status2, x, y
integer(4) ifont
character(11) face(nfonts), options(nfonts)
character(20) list
type (xycoord) xy
type (fontinfo) fi
data face /"courier", "helvetica", "times roman",    &
          "modern", "script", "roman" /
data options /"t'courier'", "t'helv'","t'tms rmn'",    &
             "t'modern'","t'script'", "t'roman'" /
! Инициализация шрифтов
if(initializefonts() .lt. 0) stop 'Шрифты не найдены'
status2 = setbkcolorrgb(#ffffff)  ! Цвет фона - ярко-белый
! Вывод имени шрифта в различных ориентациях
do ifont = 1, nfonts
! Строка опций для выбора шрифта
 list = trim(options(ifont)) // 'h30w24b'
 call clearscreen( $gclearscreen )
! Если установлен текущий шрифт
 if( setfont( list ) .ge. 0 ) then
! разместим текст по центру видеоокна. Найдем его длину и высоту
 x = (800 - getgtextextent(face(ifont)) / 2) / 2
 if( getfontinfo( fi ) .ne. 0 ) then
  call outtext( 'Нет данных о шрифте' )
  read *                            ! Ждем нажатия Enter
  cycle
 end if
  y = (600 + fi.ascent) / 2
  call moveto( x, y, xy )
  status2 = setcolor( ifont )
  call shotext ( 0 )                ! Горизонтальный текст
  call shotext ( 900 )              ! Вертикальный текст
```

```
  call shotext ( 1800 )              ! Повернутый на 180° текст
  call shotext ( 2700 )              ! Поворот на 270°
 else
  call outtext( 'Шрифт не найден' )
 end if
 read *                              ! Ждем нажатия Enter
end do
contains
 subroutine shotext(degrees)
  integer(4) degrees
  call setgtextrotation( degrees )
  call moveto( x, y, xy )            ! Позиция вывода текста в
  call outgtext(face(ifont))         ! каждой ориентации одна
 end subroutine                      ! и та же
end
```

12.12. Управление типом линий

По умолчанию графические примитивы изображаются сплошными линиями. Однако при необходимости тип линии можно изменить, применив подпрограмму

CALL SETLINESTYLE (*mask*)

mask - маска, задающая тип линии (параметр типа INTEGER(2)).

Маска представляет собой 16-битовое число, в котором каждый бит отображает пиксель линии. Если бит содержит 1, то соответствующий пиксель линии закрашивается цветом, определяемым текущим цветом и установленным способом вывода линий. Последнее выполняется функцией SETWRITEMODE. Если бит содержит 0, то соответствующий пиксель линии не изменяется. Образец воспроизводится по всей длине линии.

Устанавливаемая по умолчанию маска равна #FFFF, что соответствует сплошной линии.

Подпрограмма SETLINESTYLE оказывает влияние на способ отображения отрезков прямых, прямоугольников и многоугольников (функции LINETO, LINETO_W, POLYGON, POLYGON_W, RECTANGLE, RECTANGLE_W) и не влияет на работу функций ARC, ARC_W, ELLIPSE, ELLIPSE_W и PIE, PIE_W.

Пример. На белом фоне нарисовать параллельные линии разных типов: сплошную, из длинных и коротких штрихов, из точек.

Используем для задания типа линии шестнадцатеричное представление констант. Так, для перечисленных типов значения констант таковы (в скобках даны двоичные представления констант):

```
#FFFF - сплошная линия     (2#1111111111111111);
#FF00 - длинные штрихи     (2#1111111100000000);
#F0F0 - короткие штрихи    (2#1111000011110000);
#CCCC - точки              (2#1100110011001100).
```

```
use msflib                          ! Режим 800*600, 16 цветов
integer(2) k, dy /75/, status2
integer(2) ltype(4) /#FFFF, #FF00, #F0F0, #CCCC/
```

```
integer(4)  status4
status4 = setbkcolor(15)              ! Белый фон
call clearscreen($gclearscreen)
do k = 1, 4                           ! Вывод линий
 call drawline( )
enddo

contains
 subroutine drawline( )
  type (xycoord) xy
  status2 = setcolor(k)               ! Текущий цвет
  call setlinestyle(ltype(k))         ! Текущий тип линии
  call moveto(100, int2(100 + dy*k), xy)                    ! Начало отрезка
  status2 = lineto(700, 100 + dy*k)                         ! Рисуем отрезок
 end subroutine
end
```

Линии, как, впрочем, и иные графические примитивы, могут выводиться поверх существующего изображения. Однако для линий, прямоугольников и многоугольников можно регулировать способ их взаимодействия с существующим изображением. Это выполняется функцией

SETWRITEMODE (*wmode*)

wmode - параметр типа INTEGER(2), задающий способ взаимодействия с существующим изображением.

Функция при успешном выполнении возвращает значение типа INTEGER(2), равное величине предыдущего способа, или -1 при неудаче. Функция SETWRITEMODE оказывает влияние на работу функций LINETO, POLYGON и RECTANGLE.

Возможные значения параметра *wmode*: $GAND, $GOR, $GPRESET, $GPSET и $GXOR. Эти именованные константы определены в модуле MSFLIB. Их смысл рассмотрен в разд. 12.14. Здесь же ограничимся описанием наиболее часто используемых констант.

$GPRESET - существующее изображение замещается выводимым; при этом цвет каждого обновляемого пикселя инвертируется. Так, при выводе фиолетового пикселя его цвет после инверсии станет зеленым.

$GPSET - существующий цвет пикселей замещается цветом пикселей выводимой линии без каких-либо преобразований цвета.

По умолчанию устанавливается последний способ вывода линий.

Функция типа INTEGER(2)

GETLINESTYLE ()

возвращает значение маски для текущего типа линии.

12.13. Заполнение замкнутых областей

Замкнутые области могут быть образованы одним графическим примитивом (прямоугольник, эллипс, сектор и многоугольник) или сочетанием нескольких, например дуга и отрезки прямых.

Мы уже видели, что графические примитивы: прямоугольник, многоугольник, эллипс и сектор-могут быть заполнены текущим цветом в процессе их построения, если управляющий заполнением параметр *control*

равен $GFILLINTERIOR. При этом цвет заливки и границы примитива совпадают. Помимо этой возможности, любую замкнутую область, а не только отдельный примитив, можно заполнить текущим, установленным функцией SETCOLOR цветом, применив RGB или не RGB-функции

FLOODFILLRGB (*x, y, color*)

или

FLOODFILLRGB_W (*wx, wy, color*)

или

FLOODFILL (*x, y, bcolor*)

или

FLOODFILL_W (*wx, wy, bcolor*)

x, y - параметры типа INTEGER(2), задающие видовые координаты стартовой точки заполнения.

wx, wy - параметры типа REAL(8), задающие оконные координаты стартовой точки заполнения.

color - параметр типа INTEGER(4), задающий RGB-цвет границы.

bcolor - параметр типа INTEGER(2), задающий номер цвета границы.

Замечание. Стартовая точка заполнения также называется *затравочной точкой*, а используемый алгоритм - *алгоритмом заполнения области с затравкой.*

Функция возвращает отличное от нуля значение при успешном выполнении или 0 в противном случае. Последнее происходит, если заполнение не может быть завершено, например если стартовый пиксель (пиксель, в котором расположена стартовая точка) имеет цвет *color* (*bcolor*) или если стартовая точка лежит за пределами окна вывода. Возвращаемое функцией FLOODFILL значение имеет тип INTEGER(2), а функцией FLOODFILLRGB - INTEGER(4).

При заполнении области используются текущий цвет и текущая маска заполнения (задается подпрограммой SETFILLMASK).

Если стартовая точка находится внутри области, то заполняется сама область. Если стартовая точка находится вне области, то заполняется пространство вне области. При задании стартовой точки на границе области заполнение не производится.

Заполнение осуществляется во всех направлениях, прекращаясь при достижении пикселей, окрашенных в цвет *color* (*bcolor*) - цвет границы области.

Замечание. Заполнение области будет выполнено успешно, если область не имеет разрывов и для отображения ее границ использован сплошной тип линии.

Пример. Выполнить заливку области, расположенной между прямоугольником и размещенным внутри его эллипсом. Такая задача может быть решена с применением функций FLOODFILL или FLOODFILLRGB, если границы эллипса и прямоугольника имеют оди-

наковый цвет и стартовая точка расположена между этими границами. Заливку области и границы фигур выполним разным цветом.

```
use msflib                                          ! Режим 800*600
integer(2) status2
integer(4) status4
status4 = setbkcolorrgb(#ff0000)                    ! Цвет фона - синий
call clearscreen($gclearscreen)                     ! Окраска экрана в цвет фона
status2 = setcolorrgb(#00ffff)                      ! Желтый цвет
status2 = ellipse($gborder, 300, 250, 500, 350)
status2 = rectangle($gborder, 200, 200, 600, 400)
status2 = setcolorrgb(#008000)                      ! Светло-зеленый цвет
status2 = floodfillrgb(201, 201, #00ffff)           ! Граница желтого цвета
end
```

По умолчанию в текущий цвет окрашиваются все пиксели области. Однако можно задать образец (шаблон) заливки, который в результате применения функций FLOODFILL или FLOODFILLRGB будет воспроизведен по всей области заполнения.

Задание шаблона выполняется подпрограммой

CALL SETFILLMASK (*mask*)

mask - массив битов шаблона типа INTEGER(1).

Массив содержит 8 целых чисел. Каждый бит числа представляет пиксель экрана. Фактически при задании шаблона мы имеем дело с массивов 8 * 8 бит, пример которого приведен в табл. 12.4.

Таблица 12.4. Пример шаблона заполнения

Номер элемента массива	*Представление элемента массива*	
	двоичное	*шестнадцатеричное*
1	1 0 0 1 0 0 1 1	#93
2	1 1 0 0 1 0 0 1	#C9
3	0 1 1 0 0 1 0 0	#64
4	1 0 1 1 0 0 1 0	#B2
5	0 1 0 1 1 0 0 1	#59
6	0 0 1 0 1 1 0 0	#2C
7	1 0 0 1 0 1 1 0	#96
8	0 1 0 0 1 0 1 1	#4B

В программе при задании может быть использовано любое представление шаблона: двоичное, шестнадцатеричное, десятичное...

Наличие в бите единицы приведет к заполнению соответствующего пикселя текущим цветом. Бит с нулем оставит пиксель без изменения.

Пример. Выполнить штриховку области между эллипсом и прямоугольником.

Для штриховки используем матрицу битов, имеющую единицы на главной и двух соседних побочных диагоналях.

```
use msflib                                          ! Режим 800*600, 16 цветов
```

```
integer(1) hatch(8) / 2#00000111, 2#00001110, 2#00011100,  &
                      2#00111000, 2#01110000, 2#11100000,  &
                      2#11000001, 2#10000011 /
integer(2) status2
integer(4) status4

status4 = setbkcolor(15)            ! Цвет фона - ярко-белый
call clearscreen($gclearscreen)     ! Окраска экрана в цвет фона
status2 = setcolor(6)               ! Коричневый цвет
status2 = ellipse($gborder, 300, 250, 500, 350)
status2 = rectangle($gborder, 200, 200, 600, 400)
status2 = setcolor(7)               ! Светло-серый цвет
call setfillmask(hatch)             ! Маска-штриховка
status2 = floodfill(201, 201, 6)    ! Штриховка области

end
```

Замечание. Если совпадают цвет границы области и текущий цвет заполнения и если применяется шаблон, имеющий нулевые биты, то могут возникать ошибки заполнения.

12.14. Передача образов

Под *образом* будем понимать ограниченное прямоугольной областью изображение.

Образ может быть сохранен в оперативной или во внешней памяти, а также загружен из памяти и размещен в текущем окне вывода.

12.14.1. Обмен с оперативной памятью

Запись образа в оперативную память выполняется подпрограммами

CALL GETIMAGE (*x*1, *y*1, *x*2, *y*2, *image*)

или

CALL GETIMAGE_W (*wx*1, *wy*1, *wx*2, *wy*2, *image*)

*x*1, *y*1 - координаты левого верхнего угла ограничивающего образ прямоугольника.

*x*2, *y*2 - координаты правого нижнего угла ограничивающего образ прямоугольника.

Тип параметров *x*1, *y*1, *x*2, *y*2 - INTEGER(2).

*wx*1, *wy*1, *wx*2, *wy*2 - имеют тот же смысл. Тип параметров - REAL(8).

image - массив типа INTEGER(1), в котором сохраняется образ.

Подпрограмма сохраняет ограниченный прямоугольником образ в массиве-буфере *image*. GETIMAGE используется в видовой системе координат, а GETIMAGE_W - в оконной. Буфер должен иметь достаточные для хранения образа размеры. Размеры образа возвращаются функцией IMAGESIZE или вычисляются по формуле, приведенной при описании этой функции.

Замечание. Образ сохраняется в одномерном, типа INTEGER(1), массиве - буфере. Буфер может быть задан в программе как массив-ссылка или как размещаемый массив.

Функции

IMAGESIZE (*x*1, *y*1, *x*2, *y*2)

или

IMAGESIZE_W (*wx*1, *wy*1, *wx*2, *wy*2)

возвращают число байт, необходимых для сохранения образа в памяти.

*x*1, *y*1 - координаты левого верхнего угла образа.

*x*2, *y*2 - координаты правого нижнего угла образа.

Тип параметров *x*1, *y*1, *x*2, *y*2 - INTEGER(2).

*wx*1, *wy*1, *wx*2, *wy*2 - имеют тот же смысл. Тип параметров - REAL(8).

Тип функций - INTEGER(4).

Функция IMAGESIZE используется в видовой системе координат, а IMAGESIZE_W - в оконной. В видовой системе координат размер *size* образа определяется по формулам:

```
xwid = abs(x1 - x2) + 1
ywid = abs(y1 - y2) + 1
size = 4 + int ((xwid*bits_per_pixel + 7)/8) * ywid
```

Значение *bits_per_pixel* возвращается функцией GETWINDOWCONFIG как *bitsperpixel* (разд. 12.5).

Загрузка и размещение в видеоокне ранее сохраненного в оперативной памяти образа выполняется подпрограммами

CALL PUTIMAGE (*x*, *y*, *image*, *action*)

или

CALL PUTIMAGE_W (*wx*, *wy*, *image*, *action*)

x, *y* - параметры типа INTEGER(2), задающие видовые координаты левого верхнего угла образа.

wx, *wy* - параметры типа REAL(8), задающие оконные координаты левого верхнего угла образа.

image - массив типа INTEGER(1), содержащий образ.

action - параметр типа INTEGER(2), задающий вид взаимодействия с существующим на экране изображением.

Параметр *action* может принимать значение одной из определенных в модуле MSFLIB именованных констант:

$GAND - результирующий образ является логическим И двух образов: совпадающие точки существующего и передаваемого образов, имеющие одинаковый цвет, этот цвет сохраняют; точки, имеющие разный цвет, объединяются посредством логического И. Например, если пиксель экрана имеет бирюзовый цвет, а соответствующий пиксель образа - фиолетовый, то результирующий пиксель будет синего цвета. Это становится очевидным при рассмотрении RGB-битов названных цветов:

```
      11111111 11111111 00000000      (Бирюзовый -      #FFFF00)
AND   11111111 00000000 11111111      (Фиолетовый -     #FF00FF)
  =   11111111 00000000 00000000      (Синий -          #FF0000)
```

$GOR - загружаемый образ накладывается на существующий. Продолжая пример, получим при наложении бирюзового и фиолетового белый цвет:

```
      11111111 11111111 00000000      (Бирюзовый -      #FFFF00)
OR    11111111 00000000 11111111      (Фиолетовый -     #FF00FF)
  =   11111111 11111111 11111111      (Белый -          #FFFFFF)
```

$GPRESET - существующий образ замещается загружаемым; при этом цвет каждого загружаемого пикселя инвертируется. Так, при загрузке фиолетового пикселя его цвет после инверсии станет зеленым:

```
             11111111 00000000 11111111  (Фиолетовый -     #FF00FF)
ИНВЕРСИЯ     00000000 11111111 00000000  (Зеленый -        #00FF00)
```

$GPSET - существующий образ замещается загружаемым без каких-либо преобразований цвета.

$GXOR - обеспечивает следующее: когда образ дважды накладывается на изображение, то оно остается неизменным. Это позволяет перемещать образ, не изменяя изображения (режим $GXOR часто используется в программах анимации):

```
      11111111 11111111 00000000      (Бирюзовый -      #FFFF00)
XOR   11111111 00000000 11111111      (Фиолетовый -     #FF00FF)
  =   00000000 11111111 11111111      (Желтый -         #00FFFF)
XOR   11111111 00000000 11111111      (Фиолетовый -     #FF00FF)
  =   11111111 11111111 00000000      (Бирюзовый -      #FFFF00)
```

Замечания:

1. Если вставляемый образ выходит за пределы окна вывода, то он не отображается и экран остается без изменений.

2. В рассмотренных режимах вставки образа взаимодействуют цвета, которые пиксели получили в результате изображения графических примитивов. Пиксели, окрашенные в цвет фона, во взаимодействии не участвуют.

Пример 1. Проиллюстрировать все режимы загрузки образа при загрузке фиолетового образа в видеоокно бирюзового цвета.

```
use msflib                          ! Режим 800*600, 16 цветов
integer(1), allocatable :: buffer(:)
integer(2) k, status2
! ix, iy - размер прямоугольника, ограничивающего образ
integer(2) :: ix = 160, iy = 120
integer(2) alist(5) /$gand, $gor, $gpreset, $gpset, $gxor/
integer(4) imsize
imsize = imagesize(0, 0, ix, iy)    ! Размер образа
allocate (buffer(imsize))           ! Размещение массива
status4 = setcolorrgb(#ff00ff)      ! Цвет образа - фиолетовый
status2 = rectangle($gfillinterior, 0, 0, ix, iy)      ! Образ
call getimage(0, 0, ix, iy, buffer)  ! Сохраняем образ
status4 = setcolorrgb(#ffff00)      ! Цвет видеоокна - бирюзовый
status2 = rectangle($gfillinterior, 0, 0, 799, 599)
```

```
do k = 1, 5                          ! Вставка образа в бирюзовый экран
 call putimage(ix*(k-1), iy*(k-1), buffer, alist(k))
 read(*, *)                          ! Ждем нажатия Enter
 if(k == 5) then                     ! Повторная вставка образа в режиме $gxor
  call putimage(ix*(k-1), iy*(k-1), buffer, alist(k))
  read(*, *)                         ! Ждем нажатия Enter
 endif
enddo
deallocate(buffer)
end
```

Пример 2. Выполнить перемещение желтого шара по синусоиде, сохраняя изображение синусоиды.

```
use msflib                           ! Режим 800*600
integer(1), allocatable :: buffer(:)
! ix, iy - размер прямоугольника, ограничивающего образ
integer(2) status2, ix/40/, iy/40/, ri, xi, yi
integer(2) :: XE = 800               ! Размер видеоокна по оси x
integer(4) status4, imsize
real(8), parameter :: pi = 3.14159265_8
real(8) dx, x, y
logical(2) finv /.true./
type (xycoord) s                     ! Подготовка образа
imsize = imagesize(0, 0, ix, iy)     ! Размер образа
allocate (buffer(imsize))            ! Размещение массива
status2 = setcolorrgb(#008080)       ! Цвет шара - светло-желтый
status2 = ellipse($gfillinterior, 0, 0, ix, iy)        ! Образ
call getimage(0, 0, ix, iy, buffer)  ! Сохраняем образ
status4 = setbkcolorrgb(#c0c0c0)! Цвет фона - серый
call clearscreen($gclearscreen)      ! Очистка видеоокна
! Оконная система координат (ОСК). Рисуем в ней синусоиду
status2 = setwindow (finv, -2.0_8*pi, -2.0_8, 2.0_8*pi, 2.0_8)
dx = pi / dble(XE/2)                 ! Шаг по оси x
do x = -pi, pi, dx                   ! Изменение x в ОСК
 y = dsin(x)                         ! Значение y в ОСК
 status4 = setpixelrgb_w(x, y, #ff0000)          ! Вывод пикселя в ОСК
enddo                                ! синим цветом
! Перемещаем образ по синусоиде
ri = ix / 2                          ! Радиус шарика
do x = -pi, pi, dx                   ! Изменение x в ОСК
 y = dsin(x)                         ! Значение y в ОСК
 call getviewcoord_w (x, y, s)       ! Преобразование координат
 xi = s.xcoord - ri                  ! xi, yi - координаты вставки образа
 yi = s.ycoord - ri                  ! в видовой системе координат
 call putimage(xi, yi, buffer, $gxor)! Вставка образа
 call sleepqq(5)                                ! Задержка 5 мс
 call putimage(xi, yi, buffer, $gxor)! Восстановим изображение
enddo
deallocate(buffer)
end
```

Пояснения:

1. Задержка обеспечивается подпрограммой SLEEPQQ модуля MSFLIB.

2. Сохранение синусоиды обеспечивается режимом $GXOR и двукратной вставкой образа в одну и ту же точку видового порта.

12.14.2. Обмен с внешней памятью

Сохранение образа во внешней памяти выполняется функциями

SAVEIMAGE (*filename*, *ulxcoord*, *ulycoord*, *lrxcoord*, *lrycoord*)

или

SAVEIMAGE_W (*filename*, *ulwxcoord*, *ulwycoord*, *lrwxcoord*, *lrwycoord*)

filename - символьное выражение, содержащее полное имя файла, в который сохраняется образ.

ulxcoord, *ulycoord* - параметры типа INTEGER(4), задающие видовые координаты левого верхнего угла сохраняемого образа.

lrxcoord, *lrycoord* - параметры типа INTEGER(4), задающие видовые координаты правого нижнего угла сохраняемого образа.

ulwxcoord, *ulwycoord* - параметры типа REAL(8), задающие оконные координаты левого верхнего угла сохраняемого образа.

lrwxcoord, *lrwycoord* - параметры типа REAL(8), задающие оконные координаты правого нижнего угла сохраняемого образа.

Функция возвращает значение типа INTEGER(4), равное нулю при успешном выполнении и меньшее нуля при неудаче.

Функция сохраняет образ в Windows *bitmap*-формате (растровый формат Windows). Образ сохраняется с палитрой, содержащей выведенные на экран цвета. Используемое расширение - BMP.

Загрузка образа выполняется функциями

LOADIMAGE (*filename*, *xcoord*, *ycoord*)

или

LOADIMAGE_W (*filename*, *wxcoord*, *wycoord*)

filename - символьное выражение, задающее имя *bitmap* файла.

xcoord, *ycoord* - параметры типа INTEGER(4), задающие видовые координаты левого верхнего угла загружаемого образа.

wxcoord, wycoord - параметры типа REAL(8), задающие оконные координаты левого верхнего угла загружаемого образа.

Функция возвращает значение типа INTEGER(4), равное нулю при успешном выполнении или меньшее нуля при неудаче.

Загруженный образ отображается с использованием цветов, сохраненных в *bitmap*-файле. Следовательно, если сохраненная в *bitmap*-файле цветовая палитра отличается от текущей палитры, то новая цветовая палитра загружается вместо текущей.

Если образ выходит за пределы окна вывода, то он не загружается и изображение не изменяется.

Замечание. Образ сохраняется вместе с характеристиками используемого видеорежима. Загрузку BMP-файла образа следует выполнять в том же видеорежиме, в котором образ был создан.

Windows *bitmap*-файлы могут быть использованы в качестве фона в создаваемых в среде Фортрана изображений.

Пример. Создать последовательно фон из файлов *chitz.bmp, leaves.bmp, honey.bmp*. Приведенные файлы поставляются с Windows и находятся, как правило, в директории *c:\windows*. Для создания фона необходимо знать ширину и высоту образа в пикселях. Зная эти параметры, легко определить, сколько раз нужно повторить загрузку образа и координаты каждой загрузки. Структура BMP-файла такова: сначала следует заголовок файла (14 байт), затем заголовок изображения файла (40 байт), затем карта цветов и, наконец, растровые данные. Параметры рисунка (ширина, высота...) записаны в заголовке изображения BMP-файла (второе и третье поле заголовка изображения: байты 19-26, начиная от начала файла).

```
use msflib                          ! Режим 800*600
integer(2) :: XE = 800, YE = 600
character(50) path /'c:\windows\'/, fname, ch*18
character(12) fn(3) /'chitz.bmp', 'leaves.bmp', 'honey.bmp'/
integer(4) ix, iy, x, y, status4, i, j, k, nx, ny
do k = 1, 3                         ! Демонстрация фонов
 fname = trim(path) // fn(k)
 open(1, file = fname, form = 'binary')
 read(1) ch                         ! Пропускаем первые 18 байт файла
 read(1) ix                         ! Ширина рисунка в пикселях
 read(1) iy                         ! Высота рисунка в пикселях
 close(1)                           ! Закрываем файл для
                                    ! обеспечения его загрузки
 nx = XE / ix + 1                   ! Число вставок рисунка по x и y
 ny = YE / iy + 1
 y = 0
 do i = 1, ny
  x = 0
  do j = 1, nx
   status4 = loadimage(fname, x, y)
   x = min(x + ix, XE - ix)         ! Образ не выходит за
  enddo                             ! границу видеоокна по оси x
  y = min(y + iy, YE - iy)          ! Образ не выходит за
 enddo                              ! границу видеоокна по оси y
 read(*, *)                         ! Ожидаем нажатия Enter
enddo
end
```

12.15. Многооконный графический вывод

12.15.1. Создание дочернего окна

В приложениях QuickWin по умолчанию графический вывод направляется на окно, подсоединенное к стандартным устройствам 0 и 6. При вводе (выполняется оператором READ) открываемое по умолчанию окно ассоциируется с устройствами * и 5.

При необходимости в QuickWin можно выполнить графический вывод на дочерние окна, подсоединенные к другим устройствам. Подсоединение дочернего окна к устройству В/В выполняется оператором OPEN, в котором параметр *file* вычисляется со значением 'user'. При этом оператор OPEN может содержать дополнительную опцию TITLE = *title*, где *title* - символьное выражение, задающее заголовок открываемого окна, например:

```
open(1, file = 'user', title = 'Child window')
open(2, file = 'user', title = 'One more child window')
```

Дочернее окно может быть активным, и тогда на него направляется графический вывод. Оно может быть расположено поверх других дочерних окон. В этом случае говорят, что дочернее окно расположено в фокусе. Окно можно переместить в фокус, выполнив функцию FOCUSQQ, щелкнув по нему мышью или выполнив на него (с него) неграфический вывод (ввод) посредством PRINT, WRITE, OUTTEXT или READ. Правда, в случае неграфической передачи данных окно устанавливается в фокус, если оно открыто с IOFOCUS=.TRUE. (разд. 11.7), например:

```
open(unit = 10, file = 'user', iofocus = .true.)
```

По умолчанию IOFOCUS=.TRUE., за исключением дочерних окон, присоединяемых к устройству *. Графический вывод, например, посредством OUTGTEXT или ARC фокус не перемещает.

Все дочерние окна вывода в QuickWin располагаются в *обрамляющем* окне.

12.15.2. Размеры и положение окна

Размеры и положение окна можно задать функцией

SETWSIZEQQ (*unit, winfo*)

unit - выражение типа INTEGER(4), задающее устройство, к которому подсоединено окно. По умолчанию окно подсоединено к устройствам 0, 5 и 6. Для установки размеров обрамляющего окна в качестве номера устройства следует использовать определенную в модуле MSFLIB именованную константу QWIN$FRAMEWINDOW.

winfo - переменная производного типа qwinfo, задающая физические координаты левого верхнего угла окна, его высоту и ширину. Призводный тип *qwinfo* определен в модуле MSFLIB:

```
TYPE QWINFO
 INTEGER(2) type          ! Тип окна
 INTEGER(2) x             ! x-координаты верхнего левого угла окна
 INTEGER(2) y             ! y-координаты верхнего левого угла окна
 INTEGER(2) h             ! Высота окна
 INTEGER(2) w             ! Ширина окна
END TYPE QWINFO
```

Действие функции зависит от значения компонента *type*, которое может быть:

- QWIN$MIN - выполняется минимизация окна;
- QWIN$MAX - выполняется максимизация окна;
- QWIN$RESTORE - минимизированное окно возвращается к прежним размерам и позиции;
- QWIN$SET - положение окна и его размеры устанавливаются по значениям других компонентов *qwinfo*.

Функция возвращает значение типа INTEGER(4), равное нулю в случае успеха и отличное от нуля при неудаче.

Замечание. Позиция и размеры обрамляющего окна задаются в пикселях экрана. Позиция и размеры дочернего окна задаются в единицах высоты и ширины неграфической литеры - знакоместа (символа, выводимого операторами PRINT, WRITE и подпрограммой OUTTEXT).

Позиция и размеры окна возвращаются функцией

GETWSIZEQQ (*unit, ireq, winfo*)

Параметры *unit* и *winfo* имеют тот же смысл, что и одноименные параметры функции SETWSIZEQQ.

ireq - выражение типа INTEGER(4), задающее тип запрашиваемой информации, может принимать значение одной из определенных в модуле MSFLIB именованных констант:

- QWIN$SIZEMAX - запрос о максимально возможных размерах окна;
- QWIN$SIZECURR - получение данных о текущих размерах окна.

Функция возвращает значение типа INTEGER(4), равное нулю в случае успеха и отличное от нуля при неудаче.

Пример.

```
use msflib
logical(4) res
integer(2) numfonts, fontnum
type (qwinfo) qw
type (xycoord) pos
qw.type = qwin$max                ! Максимизация обрамляющего окна
res = setwsizeqq(qwin$framewindow, qw)
numfonts = initializefonts( )
fontnum = setfont ('t"arial"h50w34i')
data qw.x, qw.y, qw.h, qw.w / 5, 5, 10, 60 /
qw.type = qwin$set
! Задаваемое окно позволяет вывести на строке до 60 неграфических литер
result = setwsizeqq(0, qw)        ! Окно подсоединено к устройствам 0, 6 и 5
call moveto (int2(5), int2(30), pos)
call outgtext('Child window')
end
```

12.15.3. Активизация и фокусировка окна

Графический вывод и неграфический В/В выполняются на активное дочернее окно. По умолчанию активным является окно, подсоединенное к устройствам 0, 5 и 6. Оно же находится в фокусе.

Дочернее окно *unit* становится активным после выполнения QuickWin-функции

SETACTIVEQQ (*unit*)

unit - выражение типа INTEGER(4), задающее номер активного окна.

Функция возвращает значение типа INTEGER(4), равное единице в случае успеха или нулю при неудаче.

При активизации на дочернее окно можно направлять графический вывод. Графический вывод выполняется на активное окно независимо от того, находится оно в фокусе или нет. Фокус при выполнении графиче-

ского вывода не перемещается. Чтобы поместить окно в фокус, следует выполнить QuickWin-функцию FOCUSQQ, или щелкнуть по окну мышью, или направить на него неграфический В/В (посредством операторов READ, WRITE, PRINT или подпрограммы OUTTEXT). В случае неграфического В/В окно размещается в фокусе, если опция *iofocus* оператора OPEN равна .TRUE..

Установка фокуса на дочернее окно выполняется функцией

FOCUSQQ (*unit*)

unit - выражение типа INTEGER(4), задающее номер окна, которое нужно разместить в фокусе.

Функция возвращает значение типа INTEGER(4), равное нулю в случае успеха или отличное от нуля при неудаче.

После выполнения FOCUSQQ окно *unit* располагается поверх других окон. Однако при этом графический вывод может направляться на другое окно, которое стало активным после выполнения функции SETACTIVEQQ.

Номер находящегося в фокусе окна возвращается функцией

INQFOCUSQQ (*unit*)

unit - переменная типа INTEGER(4), возвращающая номер находящегося в фокусе окна.

Функция возвращает значение типа INTEGER(4), равное нулю в случае успеха или отличное от нуля при неудаче, например если находящееся в фокусе окно ассоциируется с закрытым устройством.

Пример. Вывести на двух расположенных рядом окнах проекции куба, используя два вида параллельного проецирования: изометрию и диметрию.

Пусть первоначально куб ориентирован относительно плоскости проекций так, как это показано на рис. 12.6, *а*. На экране проекцией куба при такой его ориентации будет квадрат (рис. 12.6, *б*). Такая проекция называется ортографической. Изометрия получается в результате двух поворотов куба: относительно оси *y* на угол ψ ($sin^2\psi = 1/2$), а затем относительно оси *x* на угол φ ($sin^2\varphi = 1/3$) (рис. 12.6, *в*). Оба поворота выполняются против часовой стрелки.

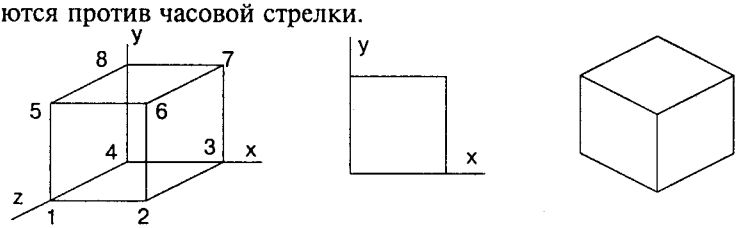

Рис. 12.6. Пространственное расположение куба и его проекции на плоскость xoy

В случае диметрии углы поворота связаны соотношением

$\sin^2\psi = \tan^2\phi$.

Новые координаты точки x^*, y^*, z^* найдем по известным старым (до поворота) координатам:

$$\begin{pmatrix} x^* \\ y^* \\ z^* \end{pmatrix} = \begin{pmatrix} 1 & 0 & 0 \\ 0 & \cos\varphi & -\sin\varphi \\ 0 & \sin\varphi & \cos\varphi \end{pmatrix} \begin{pmatrix} \cos\psi & 0 & \sin\psi \\ 0 & 1 & 0 \\ -\sin\psi & 0 & \cos\psi \end{pmatrix} \begin{pmatrix} x \\ y \\ z \end{pmatrix}$$

После преобразований получим для интересующих нас x^* и y^* координат:

x* = x * cosψ + z * sinψ
y* = x * sinφ * sinψ + y * cosφ - z * sinφ * cosψ

Осуществим вывод изометрической проекции в окно 1, а диметрии - в окно 2.

```
use msflib
! массивы x, y и z координат приведенного на рис. 12.6, а куба
real(8) :: cvex(8) = (/ 0,   100,  100,    0,    0,   100,  100,    0    /)
real(8) :: cvey(8) = (/ 0,     0,    0,    0,  100,   100,  100,  100    /)
real(8) :: cvez(8) = (/100, 100,     0,    0,  100,   100,    0,    0    /)

! массивы координат вершин проекции куба
real(8)     cvex2(8), cvey2(8), fi, psi
! массив соединяемых вершин
integer(2) inve(4, 4) / 1,    2,    4,    5,    &
                        3,    0,    0,    0,    &
                        6,    5,    7,    2,    &
                        8,    5,    7,    4 /

integer(2) :: i2                          ! win1 - окно для изометрии
integer(4) :: i4, win1 = 1, win2 = 2      ! win2 - окно для диметрии
call owi(win1, 'Cube1'c)                  ! Открываем окно win1
call owi(win2, 'Cube2'c)                  ! Открываем окно win2
fi =  dasin(1.0_8 / dsqrt(3.0_8))         ! Углы ф и ψ для изометрии
psi = dasin(1.0_8 / dsqrt(2.0_8))         ! Построение изометрии
i4 = setactiveqq(win1)                    ! Направим вывод на окно 1
call cube(cvex, cvey, fi, psi)            ! Вывод куба
fi =  dasin(1.0_8) / 4.0_8                ! Углы ф и ψ для диметрии
psi = dasin(dtan(fi))                     ! Пусть ф = π / 4
i4 = setactiveqq(win2)                    ! Направим вывод на окно 2
call cube(cvex, cvey, fi, psi)
i4 = focusqq(win1)                        ! Установим фокус на окно 1
contains
 subroutine owi(win, tit)                 ! Открываем окно win
  integer win
  character(*) tit
  type (qwinfo) qw                        ! Параметры окна
  data qw.x, qw.y, qw.h, qw.w /0, 0, 60, 60/
  qw.type = QWIN$SET
   open(win, file = 'user', title = tit)
  i4 = setwsizeqq(win, qw)
  i4 = clickmenuqq(QWIN$TILE)
  call setviewport (20, 20, 250, 250)                    ! Видовой порт для куба
  i2 = setwindow (.true._2, -100d0, -150d0, 250d0, 300d0)
  i2 = setcolor(10)                       ! Цвет ребер куба - зеленый
 end subroutine owi
```

```
 subroutine cube(cvex, cvey, fi, psi)
  real(8) cvex(:), cvey(:), fi, psi, vx, vy, dsf, dcf, dsp, dcp
  type (wxycoord) wxy
  integer(2) i, j, vi, vi2
  dsf = dsin(fi) ; dcf = dcos(fi)
  dsp = dsin(psi); dcp = dcos(psi)                    ! Координаты вершин после
  cvex2 = cvex*dcp + cvez*dsp                         ! поворота
  cvey2 = cvex*dsf*dsp + cvey*dcf - cvez*dsf*dcp
  do j = 1, 4                                         ! Вывод проекции куба
   vi = inve(1, j)
   vx = cvex2(vi); vy = cvey2(vi)
   do i = 2, 4
    vi2 = inve(i, j)
    if(vi2 == 0) cycle
    call moveto_w(vx, vy, wxy)
    i2 = lineto_w(cvex2(vi2), cvey2(vi2))
   enddo
  enddo
 end subroutine cube
end
```

Пояснение Массив *inve*(4, 4) содержит информацию о номерах соединяемых вершин. Так, первая вершина должна быть соединена с вершинами 2, 4 и 5. (Нумерация вершин приведена на рис. 12.6, *а*). Ребра, выходящие из вершины 3, невидимы. Поэтому список вершин, инцидентных вершине 3, состоит из нулей. Координаты *x*, *y* и *z* *i*-ой вершины куба до проецирования равны *cvex*(*i*), *cvey*(*i*) и *cvez*(*i*).

Функция CLICKMENUQQ позволяет имитировать выбор команды QuickWin меню. Если параметр функции принимает значение QWIN$TILE, то дочерние окна размещаются рядом друг с другом.

12.16. Статус выполнения графических процедур

Проконтролировать выполнение графических процедур можно посредством функции

GRSTATUS ()

Функция возвращает значение типа INTEGER(2), которое меньше нуля при ошибочном завершении последней использованной графической процедуры, которое больше нуля при наличии предупреждения или которое равно нулю в случае успеха. Функция GRSTATUS может быть использована сразу после вызова графической процедуры, чтобы зафиксировать возможные ошибки или предупреждения. Для нулевого значения результата функции GRSTATUS в модуле MSFLIB определена символьная константа $GROK. Также в этом модуле определены и именованные константы, которым присвоены коды ошибок и предупреждений.

Обработка ошибок графического вызова может быть выполнена, например, так:

```
if ( grstatus() < $grok ) then
! обработка ошибки выполнения графической процедуры
```

```
...
endif
```

Перечисленные ниже процедуры не приводят к ошибкам, и всегда после их применения GRSTATUS возвратит $GROK (0):

DISPLAYCURSOR, GETTEXTCOLORRGB, GETBKCOLOR,
GETTEXTPOSITION, GETBKCOLORRGB, GETTEXTWINDOW,
GETCOLOR, GETCOLORRGB, GETTEXTCOLOR,
OUTTEXT, WRAPON.

Приложение 1. Метакоманды FPS

Метакоманды - это специальные включаемые в программу инструкции, которые сообщают компилятору, какие следует предпринять действия при компиляции программы. Помимо метакоманд, процесс компиляции может управляться и из командной строки. Редактировать командную строку компилятора можно, например, в среде Microsoft Developer Studio, выбирая пункты меню Options - Project - Compiler. Однако управление метакомандами является более гибким, поскольку последние могут быть включены, затем выключены или изменены в различных местах исходного текста программы. При возникновении конфликта между опцией компилятора и метакомандой приоритет имеет метакоманда.

Воздействие метакоманды распространяется на часть кода, расположенного после появления в нем метакоманды. В большинстве случаев действие метакоманды можно отменить или изменить. Так, многие метакоманды имеют метакоманду, отменяющую их действие, например $DEBUG и $NODEBUG, или могут быть заданы с различными значениями, например $REAL:8 устанавливает задаваемый по умолчанию размер вещественного типа данных равным 8 байтам, а $REAL:4 уставливает этот размер равным 4 байтам. Таким образом, одна часть программы может быть откомпилирована согласно метакоманде $REAL:8, а другая - $REAL:4; в одной части программы может быть включен отладчик (метакоманда $DEBUG), а в другой - выключен. И так далее.

Метакоманды подразделяются на категории:

- метакоманды $STRICT, $NOFREEFORM и $FIXEDFORMLINESIZE, сообщающие компилятору о необходимости выявления в исходном коде отличий от стандарта ($STRICT) или заданого формата. Используются, если необходимо создать переносимый на другие компиляторы код. Для первых двух метакоманд существуют обратные: $NOSTRICT и $FREEFORM;
- метакоманды, обеспечивающие компиляцию фрагмента программы при выполнении некоторого условия: $DEFINE, $UNDEFINE, $IF, $IF DEFINED, $ELSE, $ELSEIF и $ENDIF;
- метакоманды, управляющие процессом отладки: $DEBUG, $NODEBUG, $DECLARE, $NODECLARE, $LINE и $MESSAGE;
- метакоманды, изменяющие задаваемый по умолчанию размер встроенных целого и вещественного типов данных: $INTEGER и $REAL;
- метакоманды, управляющие печатью листинга исходного кода: $LIST, $NOLIST, $LINESIZE, $PAGE, $PAGESIZE, $TITLE и $SUBTITLE;
- метакоманда $OPTIMIZE, управляющая опциями оптимизации компилируемого кода;
- метакоманда $OBJCOMMENT, помещающая полное имя файла библиотеки в объектный код для ее поиска компоновщиком;
- метакоманда $PACK, управляющая начальным адресом сохранения компонентов производных типов;

- метакоманда $ATTRIBUTES объявляет Microsoft-атрибуты.

Замечания:

1. Как и прежде, необязательные элементы метакоманд будут заключаться в квадратные скобки.

2. Метакоманда $INCLUDE и оператор INCLUDE поддерживаются в FPS4, поэтому нет необходимости менять существующий исходный код, хотя в FPS они могут быть полностью заменены модулями.

П.1.1. Использование метакоманд

Любая строка исходного текста, содержащая символ доллара ($) в первой позиции или начинающаяся с символов !MS$, интерпретируется как метакоманда. При задании метакоманды символы !MS$ могут начинаться в любой позиции строки. Им могут предшествовать только пробелы и символы табуляции. Буквы MS могут быть прописными, строчными или смешанными. Четыре символа !MS$ должны быть записаны без встроенных пробелов; пробелы после $ игнорируются. Например:

```
$debug
      !ms$optimize:'on:p'
$optimize:'off'
!ms$              nodebug
```

Хотя префикс !MS$ в написании метакоманд введен начиная с версии FPS4, форма метакоманды с !MS$ предпочтительнее, поскольку другие компиляторы обрабатывают строку с такой метакомандой, как комментарий, что облегчает перенос написанных в FPS программ на другие системы.

Метакоманда и ее параметры должны располагаться на одной строке программы, поскольку продолжение строки с метакомандой невозможно. Большинство метакоманд могут размещаться на любой строке исходного файла. Их действие, если они не отменены другой метакомандой, распространяется до конца файла. Это позволяет задавать разные директивы компилятору для разных фрагментов исходного кода. Однако существуют ограничения. Нельзя разместить метакоманду $OPTIMIZE внутри процедуры. Метакоманды $INTEGER, $REAL, $STRICT и $NOSTRICT могут появляться только в верхней части программной единицы: головной программе, во внешней процедуре, в модуле и BLOCK DATA.

Дествие метакоманд распространяется на любой включаемый (INCLUDE) файл, который, правда, может содержать свои собственные метакоманды. Метакоманды, расположенные внутри включаемого файла, оказывают действие как на код включаемого файла, так и на расположенный вслед за ним код файла-хозяина, то есть файла, в который выполняется вставка кода. Однако если метакоманда внутри включаемого файла изменяет формат записи исходного кода или длину строки ($FREEFORM, $NOFREEFORM или $FIXEDFORMLINESIZE), то ее действие распространяется только на включаемый файл и не затрагивает файла-хозяина.

Метакоманды программы, которая подключает модуль оператором USE, не оказывают влияния на код модуля. Поэтому для управления компиляцией модуля в него следует включить свои собственные метакоманды.

Поскольку метакоманда $PACK оказывает воздействие на распределение памяти, то ее нельзя размещать внутри управляющих конструкций (например, внутри конструкции IF - THEN - ELSE) или внутри определений производных типов данных.

Конструкция с метакомандой $IF может включать обычные операторы Фортрана. Правда, эти операторы должны быть исключены при использовании кода в других системах программирования.

Пока действует метакоманда $STRICT, требующая соблюдения стандарта Фортран 90, все другие метакоманды в форме !MS$*metacommand* интерпретируются как комментарий, не вызывая каких-либо действий или ошибок.

На одной строке вслед за метакомандой можно добавить начинающийся с восклицательного знака комментарий, например:

```
!ms$define test                          ! Задание символа test
$message:'Compiling Subroutine ERF'      ! Вывод сообщения на экран
```

П.1.2. Метакоманды, контролирующие правила написания исходного кода

П.1.2.1. Метакоманды $STRICT и $NOSTRICT

Метакоманда $STRICT указывает компилятору на необходимость выявления элементов Фортрана Microsoft, не соответствующих стандарту Фортран 90. $NOSTRICT (задается по умолчанию) позволяет использование всех имеющихся в FPS расширений.

Синтаксис метакоманд:

$STRICT или !MS$STRICT

$NOSTRICT или !MS$NOSTRICT

Метакоманды $STRICT и $NOSTRICT могут появляться только в верхней части программной единицы: головной программе, внешней процедуре, модуле или BLOCK DATA - и не могут появляться между программными единицами или в начале внутренней процедуры. Они не оказывают влияния на включаемые оператором USE модули.

Пример.

```
type stuff                                  ! $nostrict по умолчанию
  integer(4) k, m
  character(40) name
end type stuff
type (stuff) examp
double complex :: cd=(3.0d0, 4.0d0)         ! Нестандартный тип - нет ошибки
examp.k = 4                                 ! Нестандартный селектор
end                                         ! компонента - нет ошибки
```

```
subroutine strictdemo( )
 !ms$strict
 type stuff
  integer(4) k, m
  character(40) name
 end type stuff
 type (stuff) samp
 double complex :: cd=(3.0d0, 4.0d0)          ! Ошибка. Правильно complex(8)
 samp.k = 4                                   ! Ошибка. Правильно samp%k = 4
end subroutine
```

П.1.2.2. Метакоманды $FREEFORM и $NOFREEFORM

Метакоманда $FREEFORM позволяет записывать исходный код в свободном формате. Метакоманда $NOFREEFORM, наоборот, указывает на необходимость записи текста программы в фиксированном формате (см. прил. 4). Синтаксис:

$FREEFORM или !MS$FREEFORM

$NOFREEFORM или !MS$NOFREEFORM

Существует два способа установить по умолчанию свободный формат записи исходного кода:

- использовать расширение F90 для файлов с исходным кодом;
- использовать опцию компилятора /4Yf.

Если ни один из этих способов не использован, то по умолчанию устанавливается фиксированный формат записи исходного кода. Дополнительно к существующим правилам умолчания можно установить метакомандой $FREEFORM свободный, а метакомандой $NOFREEFORM фиксированный формат записи исходного кода.

Когда применена метакоманда $FREEFORM или $NOFREEFORM, она действует на всю расположенную вслед за ней часть файла или до тех пор, пока не использована отменяющая ее метакоманда. В случае применения метакоманда воздействует на INCLUDE-файлы, но не действует на USE-модули, которые компилируются отдельно. Длину строки фиксированного формата можно изменить метакомандой $FIXEDFORMLINESIZE.

Пример.

```
!ms$freeform                          ! Свободный формат записи исходного кода
integer(4) k, m
k = 4; m = 5
print 1, k, m
1 format(2i4)
call demo( )
end

    subroutine demo( )
!ms$nofreeform                        ! Фиксированный формат. Для отображения
        integer(4) k/4/, m/5/         ! в среде FPS разделяющих полос
        print 1, k, m                 ! метакоманду следует начать с первой
   1    format(2i4)                   ! позиции строки
    end
```

П.1.2.3. Метакоманда $FIXEDFORMLINESIZE

Метакоманда $FIXEDFORMLINESIZE устанавливает длину строки фиксированного формата исходного кода. Ее синтаксис:

$FIXEDFORMLINESIZE: 72 | 80 | 132

или

!MS$FIXEDFORMLINESIZE: 72 | 80 | 132

Применяя метакоманду, можно установить длину строки в фиксированном формате равной 72, 80 или 132 символа. По умолчанию длина строки составляет 72 символа. $FIXEDFORMLINESIZE действует либо до конца файла, либо до следующей, изменяющей ее метакоманды. Как и другие метакоманды, она воздействует на INCLUDE файлы, но не действует на USE модули. Если INCLUDE файл изменяет длину строки, то изменения не влияют на длину строки файла-хозяина.

Метакоманда $FIXEDFORMLINESIZE не оказывает действия на код, написанный в свободном формате.

Пример.

```
!ms$nofreeform
!ms$fixedformlinesize:132
      print *, 'Пишем эту строку за пределами 72-й колонки без продолжения'
```

П.1.3. Условная компиляция программы

П.1.3.1. Метакоманды $DEFINE и $UNDEFINE

Метакоманда $DEFINE создает символическую переменную, существование которой может быть проверено во время условной компиляции. $UNDEFINE удаляет созданную $DEFINE переменную.

Синтаксис:

$DEFINE *symbol-name* [= *val*]

или

!MS$DEFINE *symbol-name* [= *val*]

$UNDEFINE *symbol-name*

или

!MS$UNDEFINE *symbol-name*

symbol-name - имя переменной, содержащее до 31 символа; может начинаться с $ и знака подчеркивания (_) и не может начинаться с цифры.

val - присвоенное *symbol-name* значение. Тип *val* - INTEGER(4).

Метакоманды $DEFINE и $UNDEFINE создают и удаляют переменные для использования с метакомандами $IF и $IF DEFINED. Имя, определенное метакомандой $DEFINE, является локальным (используется только другими метакомандами и недоступно в программе) и поэтому может быть повторно объявлено в Фортран-программе. То есть

$DEFINE-имена могут дублировать имена объектов данных программы без каких-либо конфликтов.

Для проверки того, определен символ или нет, используется метакоманда $IF DEFINED (*symbol-name*). Для проверки присвоенного *symbol-name* значения используется метакоманда $IF. В логических выражениях метакоманды $IF могут использоваться большинство логических и арифметических операций Фортрана.

Попытка удалить метакомандой $UNDEFINE символ, который не был определен, приведет к ошибке компиляции.

Метакоманды $DEFINE и $UNDEFINE могут появляться в любом месте программы.

Пример.

```
!ms$define  tflag
real :: tflag = 4
!ms$if defined (tflag)
  write (*,*) 'Компилирую строку A'
!ms$else
  write (*,*) 'Компилирую строку B'
!ms$endif                          ! Нет конфликта между именами
print *, tflag                     !       4.000000
!ms$define  tf = 2
!ms$if (tf == 1)                   ! или (tf .eq. 1)
  write (*, *) 'Компилирую строку C'
!ms$else
  write (*, *) 'Компилирую строку D'
!ms$endif
end
```

Потребность в применении метакоманд условной компиляции возникает на этапе создания исходного кода. Например, используя символическое имя, можно иметь в исходном тексте два варианта кода для некоторого фрагмента алгоритма, компилируя тот или иной вариант кода в зависимости от значения введенного метакомандой имени. После отладки неиспользуемый код может быть удален.

Если в метакоманде $DEFINE имени присваивается целочисленное значение, то такое имя можно использовать в другой метакоманде $DEFINE для определения значения другого символического имени, например:

```
!ms$define firstsym = 100000
!ms$define receiver = firstsym     ! Имя firstsym получило
...                                ! целочисленное значение в одной из
!ms$if receiver .ne. 100000        ! предшествующих метакомад $define
     write(*, *) "Компилируем эту часть кода"
     ...
!ms$else
     write(*, *) "Компилируем другую часть кода"
     ...
!ms$endif
```

П.1.3.2. Конструкции метакоманд $IF и $IF DEFINED

Конструкции метакоманд условной компиляции $IF и $IF DEFINED используются для выполнения условной компиляции. Метакоманды условной компиляции начинаются с $IF или $IF DEFINED и заканчиваются метакомандой $ENDIF. Конструкция может включать одну или несколько метакоманд $ELSEIF и одну метакоманду $ELSE. Если некоторое логическое условие в конструкции вычисляется со значением .TRUE., а все предшествующие логические условия в конструкции $IF вычисляются со значением .FALSE., то выполняется компиляция содержащихся в блоке метакоманды операторов. Синтаксис метакоманд:

$IF [(] *expr* [)] или $IF DEFINED (*symbol_name*)
statementblock
[$ELSEIF [(] *expr* [)]
statementblock]
[$ELSE
statementblock]
$ENDIF

или

!MS$IF [(] *expr* [)] или !MS$IF DEFINED (*symbol_name*)
statementblock
[!MS$ELSEIF [(] *expr*[)]
statementblock]
[!MS$ELSE
statementblock]
!MS$ENDIF

expr - вычисляемое со значением .TRUE. или .FALSE. логическое выражение.

symbol_name - определяемое метакомандой $DEFINE или опцией компилятора /D символическое имя. В метакоманде $IF DEFINED проверяется существование символа. Удаление ранее определенного символа выполняется метакомандой $UNDEFINE.

statementblock - операторы программы, которые компилируются или не компилируются в зависимости от значения логических выражений в конструкции $IF.

$IF DEFINED (*symbol_name*) - вычисляется со значением .TRUE., если *symbol_name* определено метакомандой $DEFINE (со значением или без) и не удалено метакомандой $UNDEFINE, в противном случае $IF DEFINED (*symbol_name*) вычисляется со значением .FALSE.. Метакоманде $IF DEFINED (*symbol_name*) недоступны определенные в тексте программы имена.

Если логическое условие метакоманд $IF или $IF DEFINED есть .TRUE., то компилируются операторы следующего за этими метакомандами *statementblock*. Если условие вычисляется со значением .FALSE., то управление передается следующим метакомандам $ELSEIF или $ELSE (при наличии таковых). Таким же образом обрабатывается логическое

условие метакоманды $ELSEIF. Если управление передано метакоманде $ELSE, то выполняется следующий за метакомандой *statementblock*.

Наличие закрывающей конструкцию $IF метакоманды $ENDIF обязательно.

В логических выражениях метакоманд можно использовать любые логические операции и операции отношения Фортрана: .LT., <, .GT., >, .EQ., ==, .LE., <=, .GE., >=, .NE., /=, .EQV., .NEQV., .NOT., .AND., .OR. и .XOR.. Логические выражения могут быть любой сложности, но вся метакоманда (включая условие) должна быть размещена на одной строке. Приоритет выполнения логических операций такой же, как и в стандартных выражениях отношения и логических выражениях Фортрана. В выражениях метакоманд условной компиляции логические операции .EQV., .NEQV., .NOT., .AND., .OR. и .XOR. могут быть использованы только с логическими величинами.

Пример.

```
!ms$define flag = 3
!ms$if (flag .lt. 2)
 print *, "Компилирую этот блок, если flag < 2"
!ms$elseif (flag >= 8)
 print *, "Компилирую этот блок, если flag >= 8"
!ms$else
 print *, "Компилирую этот блок, если предшествующие условия - ложь"
!ms$endif
end
```

Условная компиляция может быть использована и в модуле. Однако нельзя использовать условную компиляцию для всего модуля в среде FPS, поскольку невозможно будет установить необходимые связи в проекте до компиляции. Например, следующий фрагмент ошибочен:

```
!ms$if defined (debug)
 module mod
  real :: b = 3.0
 end module
!ms$endif
```

А этот текст компилируется без ошибок:

```
module mod
!ms$if defined (debug)
 real :: b = 3.0
!ms$endif
end module
```

Метакоманды условной компиляции ($IF, $IF DEFINED, $ELSE, $ELSEIF и $ENDIF) упрощают процесс разработки программы, позволяя пропускать незавершенный код, использовать или пропускать код тестовых вставок, настраивать код для специальных приложений путем подключения и исключения соответствующих фрагментов.

П.1.4. Управление отладкой программы

П.1.4.1. Метакоманды $DEBUG и $NODEBUG

Метакоманда $DEBUG сообщает компилятору о необходимости дополнительных проверок и более детальной обработки ошибок. Она также может быть использована для условной компиляции. Метакоманда $NODEBUG (установлена по умолчанию) подавляет дополнительные проверки и более детальную обработку ошибок.

Синтаксис:

$DEBUG[:*string*] или $NODEBUG[:*string*]

!MS$DEBUG[:*string*] или !MS$NODEBUG[:*string*]

string - необязательный параметр типа CHARACTER(*). Если параметр задан, то компилируются только строки, содержащие букву строки *string* в первом столбце. Регистр не является значащим.

Метакоманды $DEBUG и $NODEBUG могут появляться в любом месте исходного файла. Когда отладочный режим активизирован метакомандой $DEBUG, компилятор:

- проверяет целочисленную арифметику на переполнение;
- снабжает систему обработок ошибок именами файлов и номерами строк. Если происходит ошибка выполнения, то связанные с ошибкой имя файла, номер строки и состояние стека отображаются на экране;
- контролирует границы массивов и строк. С целью уменьшения числа связанных с нарушением границ ошибок лучше использовать в процедурах массивы-параметры заданной формы и массивы, перенимающие форму;
- проверяет диапазон присваивания. Ошибки выявляются, когда, например, значение *ival* типа INTEGER(4) присваивается переменной типа INTEGER(2), причем значение *ival* превышает максимально возможную для INTEGER(2) величину. Если метакоманда $DEBUG не задействована, то выполняется отсечение значения, ошибка не генерируется и результат работы программы непредсказуем. В случае вещественного типа данных всегда при некорректном присваивании возникает ошибка.

В случае написания исходного кода в свободном формате невозможна условная компиляция, использующая букву строки *string* метакоманды $DEBUG:*string*. В фиксированном формате буква *C* (или *c*) в первой позиции всегда означает начало строки - комментария, и, следовательно, наличие *C* в *string* не дает никакого эффекта. Если задано более одной метакоманды $DEBUG:*string*, каждая строка *string* замещает предыдущую строку *string*.

Метакоманда $DEBUG не оказывает влияния на обработку ошибок вычислений с плавающей точкой. Для обработки таких ошибок используются процедуры библиотеки MATHERRQQ.

Пример.

```
!ms$nofreeform
!ms$debug:'abcd'
   integer i
a    i = 1
e    i = 2                    ! Среди отмеченных в первой позиции
b    i = i + i                ! операторов выполняются только
f    i = i * i                ! операторы a и b
c    i = i + 1                ! Строка-комментарий
   write(*, *) i              ! 2
```

П.1.4.2. Метакоманды $DECLARE и $NODECLARE

Метакоманда $DECLARE вырабатывает предупреждение для переменных, если они не появились в операторах объявления типа (работает подобно IMPLICIT NONE). $NODECLARE (устанавливается по умолчанию) отключает выработку предупреждений. Синтаксис:

$DECLARE или !MS$DECLARE

$NODECLARE или !MS$NODECLARE

Метакоманда преимущественно $DECLARE используется на этапах отладки программы для выявления необъявленных либо объявленных, но неиспользующихся переменных.

П.1.4.3. Метакоманда $MESSAGE

Метакоманда посылает символьную строку на стандартное устройство вывода (экран) во время первого прохода компилятора. Предназначена для облегчения процесса отладки. Синтаксис:

$MESSAGE:*string* или !MS$MESSAGE:*string*

string - заключенная в апострофы или двойные кавычки строка сообщений. В среде FPS сообщения выводятся в окно Build.

Пример.

```
!ms$message: 'Компиляция подпрограммы setpoint( )'
```

П.1.5. Выбор задаваемой по умолчанию разновидности типа

П.1.5.1. Метакоманда $INTEGER

Метакоманда устанавливает задаваемое по умолчанию значение параметра разновидности целого типа. Синтаксис:

$INTEGER: 2 | 4 или !MS$INTEGER: 2 | 4

Пользуясь метакомандой $INTEGER, можно установить задаваемую по умолчанию разновидность целого типа с параметрами разновидности KIND = 2 или KIND = 4. Метакоманда оказывает действие только на объекты данных, объявляемые оператором INTEGER без указания пара-

метра разновидности, и не оказывает действия на буквальные константы и на объекты, введенные без объявления типа. При отсутствии метакоманды (если не задана опция компилятора /4I2) по умолчанию устанавливается разновидность целого типа с KIND = 4.

Метакоманда $INTEGER может появляться только в верхней части программной единицы: головной программы, внешней процедуры, модуля или BLOCK DATA. $INTEGER не может появляться между программными единицами или в начале внутренней процедуры. Метакоманда не воздействует на модули, включаемые оператором USE.

Задаваемые по умолчанию разновидности логического и целого типа данных всегда совпадают. Таким образом, действие метакоманды $INTEGER распространяется и на логические объявленные оператором LOGICAL объекты данных.

Пример.

```
integer i                          ! 4-байтовая целочисленная переменная
logical fl                         ! 4-байтовая логическая переменная
k = 2                              ! 4-байтовая целочисленная переменная
print *, kind(i), kind(k)          !       4       4
print *, kind(fl), kind(.false.)   !       4       4
call integer2( )
print *, kind(i), kind(k)          !       4       4
end

subroutine integer2( )             ! $integer:2 не оказывает действия
 !ms$integer:2                     ! на вызывающую программную единицу
 integer j                         ! 2-байтовая целочисленная переменная
 logical fl                        ! 2-байтовая логическая переменная
 print *, kind(j), kind(3)         !       2       4
 print *, kind(fl), kind(.true.)   !       2       4
end subroutine
```

П.1.5.2. Метакоманда $REAL

Метакоманда устанавливает задаваемое по умолчанию значение параметра разновидности вещественного типа. Синтаксис:

$REAL: 4 | 8 или !MS$REAL: 4 | 8

Пользуясь метакомандой $REAL, можно установить задаваемую по умолчанию разновидность вещественного типа с параметрами разновидности KIND = 4 или KIND = 8. Метакоманда оказывает действие только на объекты данных, объявляемые оператором REAL без указания параметра разновидности, и не оказывает действия на буквальные константы и на объекты, введенные без объявления типа. При отсутствии метакоманды (если не задана опция компилятора /4R8) по умолчанию устанавливается разновидность вещественного типа с KIND = 4.

Метакоманда $REAL может появляться только в верхней части программной единицы: головной программы, внешней процедуры, модуля или BLOCK DATA. Метакоманда $REAL не может появляться между программными единицами или в начале внутренней процедуры. Метакоманда не воздействует на модули, включаемые оператором USE.

Пример.

```
real r                          ! 4-байтовая вещественная переменная
write(*, *) kind(r)             !        4
call real8( )
write(*, *) kind(r)             !        4
end
subroutine real8( )
 !ms$real:8
 real s                         ! 8-байтовая вещественная переменная
 write(*, *) kind(s)            !        8
end subroutine
```

П.1.6. Управление печатью листинга исходного кода

П.1.6.1. Метакоманды $LIST и $NOLIST

Метакоманда $LIST (задается по умолчанию) сообщает компилятору на необходимость вывода информации в заданный файл листинга. $NOLIST подавляет вывод листинга. Синтаксис:

$LIST или $NOLIST

!MS$LIST или !MS$NOLIST

$LIST и $NOLIST могут появляться в любом месте исходного кода.

П.1.6.2. Метакоманда $LINESIZE

Метакоманда задает число символов в строке в выводимом компилятором листинге. Синтаксис:

$LINESIZE:*n* или !MS$LINESIZE:*n*

n - целое число в диапазоне от 80 до 132, задающее число символов в строке выводимого компилятором листинга. По умолчанию $n = 80$.

$LINESIZE может появляться в любом месте исходного кода.

П.1.6.3. Метакоманда $PAGE

Метакоманда начинает новую страницу в выводимом компилятором листинге. Может появляться в любом месте исходного кода. Синтаксис:

$PAGE или !MS$PAGE

П.1.6.4. Метакоманда $PAGESIZE

Метакоманда устанавливает длину страницы в выводимом компилятором листинге. Синтаксис:

$PAGESIZE:*n* или !MS$PAGESIZE:*n*

n - новое число строк на странице листинга ($n \geq 15$).

$PAGESIZE может появляться в любом месте исходного кода. По умолчанию длина страницы листинга равна 63 строкам. При этом вместе с заголовками общая длина страницы составляет 66 строк.

П.1.6.5. Метакоманда $TITLE

Метакоманда задает вывод специального заголовка на каждой странице листинга. Заголовок выводится до тех пор, пока не изменен другой метакомандой $TITLE. Синтаксис:

$TITLE: '*title*' или !MS$TITLE: '*title*'

title - символьная буквальная константа.

$TITLE может появляться в любом месте исходного кода. Для отключения вывода заголовка следует задать метакоманду $TITLE, опция *title* которой является пустой строкой. При отсутствии метакоманды никакого заголовка не выводится.

Пример.

```
!ms$title: 'Расчет прочности кузова. Версия 2.0 11/28/98'
integer i, j, k
real a, b, c
call track(i, j, k, a, b, c)
. . .
```

П.1.6.6. Метакоманда $SUBTITLE

Метакоманда задает вывод специального подзаголовка на каждой странице листинга. Подзаголовок выводится до тех пор, пока не изменен другой метакомандой $SUBTITLE. Синтаксис:

$SUBTITLE: '*subtitle*' или !MS$SUBTITLE: '*subtitle*'

subtitle - символьная буквальная константа.

При отсутствии метакоманды никакого подзаголовка не выводится. $SUBTITLE может появляться в любом месте исходного кода. Для отключения вывода заголовка следует задать метакоманду $SUBTITLE, опция *subtitle* которой является пустой строкой.

Пример.

```
!ms$title: 'Расчет прочности кузова. Версия 2.0 11/28/98'
integer i, j, k
real a, b, c
call track (i, j, k, a, b, c)
call odat (b, c)
end
subroutine track (i, j, k, a, b, c)
  !ms$subtitle: 'Подпрограмма track'
  integer i, j, k
  real a, b, c
  call stat(a, b)
  !ms$subtitle: ''
end subroutine track
```

П.1.7. Управление опциями оптимизации исходного кода. Метакоманда $OPTIMIZE

Метакоманда управляет оптимизацией исходного кода. Синтаксис:

$OPTIMIZE: 'ON | OFF[:*optionlis*]'

или

!MS$OPTIMIZE: 'ON | OFF[:*optionlist*]'

optionlist - необязательный список опций, оптимизирующих работу компилятора. При наличии в списке более одной опции они разделяются запятыми. Ключевые слова ON или OFF и :*optionlist* задаются без пробелов. Могут быть заданы опции:

- d - отключает оптимизацию (задается по умолчанию);
- p - улучшение согласованности операций с плавающей точкой и проверок ошибок исполнения;
- x - оптимизация по скорости выполнения без проверки ошибок;
- xp - оптимизация по скорости выполнения с проверкой математических ошибок выполнения.

$OPTIMIZE изменяет опции оптимизации компилятора. Изменения действуют до конца файла либо до тех пор, пока не появилась метакоманда $OPTIMIZE с другими опциями. Метакоманда $OPTIMIZE должна появляться за пределами любой программной единицы, но может быть размещена между внешними, внутренними процедурами и модулями, расположенными в пределах одного файла.

Ключевые слова ON или OFF должны появляться в строковом аргументе метакоманды. Ключевое слово задается прописными буквами. Если используется ON, то заданные опции будут работать. При задании OFF заданные опции будут отключены. Опции могут быть заданы с переключателем /O в командной строке компилятора. Опция /Ob2 в командной строке компилятора не может быть использована с $OPTIMIZE.

Если ON применяется без *optionlist*, то восстанавливаются заданные в командной строке компилятора опции оптимизации. Если же OFF используется без *optionlist*, то все опции оптимизации отключаются (тот же эффект имеет переключатель компилятора /Od).

Как правило, опции оптимизации применяются при создании готового программного продукта. В процессе его разработки и отладки с целью экономии времени компиляция выполняется без оптимизации.

Пример.

```
!ms$optimize: 'ON:p'                    ! 'ON:p' - задается без пробелов
write(*, *) 'Компилирую код с оптимизацией по проверке ошибок'
call one( )
call two( )
end

$optimize: 'OFF'
subroutine one
 write(*, *) 'Оптимизация отключена
end subroutine one
```

```
!ms$optimize: 'ON:xp'
subroutine two
 write(*, *) 'Оптимизирую по скорости и проверяю ошибки'
end subroutine two
```

П.1.8. Метакоманда $OBJCOMMENT

Метакоманда помещает полное имя библиотеки с указанием пути в объектный файл. Синтаксис:

$OBJCOMMENT LIB: "*library*"

или

!MS$OBJCOMMENT LIB: "*library*"

library - символьная буквальная константа, содержащая имя и при необходимости путь библиотеки, которую должен использовать компоновщик.

Компоновщик ищет библиотеку, указанную в $OBJCOMMENT (тот же эффект имеет указание имени библиотеки в командной строке), прежде чем им выполняется поиск установленной по умолчанию библиотеки. Можно в одном исходном файле задать несколько метакоманд, ссылающихся на разные библиотеки. Имена библиотек, которые должен использовать компоновщик, появляются в объектном файле в порядке их введения метакомандами $OBJCOMMENT в исходном коде.

Если метакоманда $OBJCOMMENT появляется в пределах модуля, то каждая использующая модуль программная единица также включает эту метакоманду. Если необходимо включить метакоманду в модуль, но не передавать ее в использующие модуль программные единицы, то следует поместить $OBJCOMMENT перед предложением MODULE *имя_модуля*, например:

```
module a                                          ! файл mod1.f90
 !ms$objcomment lib: "opengl32.lib"
 . . .
end module a
!ms$objcomment lib: "graftools.lib"               ! файл mod2.f90
module b
 . . .
end module b

program go              ! файл user.f90
 use a                  ! Поиск библиотеки включается в объектный код
 use b                  ! Поиск библиотеки не включается в объектный код
 . . .                  ! файла user.obj
end
```

П.1.9. Метакоманда $PACK

Метакоманда $PACK управляет начальными адресами памяти компонентов производных типов. Синтаксис:

$PACK: [1 | 2 | 4] или !MS$PACK: [1 | 2 | 4]

Элементы производных типов данных (структур) и объединений (UNION) выравниваются в памяти ЭВМ двумя способами: по размеру типов элементов или в соответствии с текущими установками выравнивания. Текущие установки выравнивания могут принимать значения 1, 2, 4 или 8 байт. Начальное значение выравнивания памяти может быть выбрано равным 1, 2 или 4 байтам посредством опции компилятоа /Zp. По умолчанию начальная установка составляет 8 байт. Уменьшая значение установки выравнивания, можно более плотно паковать данные производных типов в памяти компьютера.

Метакоманда $PACK позволяет дополнительно управлять упаковкой компонентов производных типов данных внутри программы, изменяя заданную начальную установку выравнивания. Метакоманда может появляться в любом месте программы перед объявлением производного типа данных (структуры) и не может появляться внутри объявлений производных типов.

Если метакоманда $PACK не задана, то ко всей программе применяется или установленное по умолчанию значение выравнивания компонентов структур в памяти (8 байт), или значение, заданное опцией /Zp компилятора.

Задание $PACK:1 означает, что все переменные начинаются на следующем свободном байте (четном или нечетном). Следовательно, между компонентами не образуется свободной памяти, хотя это и приводит к некоторому увеличению времени доступа. Если задана метакоманда $PACK:4 INTEGER(1), LOGICAL(1) и все символьные переменные начинаются на следующем свободном байте (четном или нечетном), INTEGER(2) и LOGICAL(2) начинаются на следующем четном байте, все другие переменные начинаются на 4-байтовой границе.

Если метакоманда $PACK: задана без числа, то установка выравнивания возвращается к заданному опцией компилятора /Zp значению или к задаваемым по умолчанию 8 байтам.

Пример. Выведем расстояние в памяти между двумя компонентами структуры при различных упаковках. Типы компонентов - INTEGER(1) и INTEGER(4). Напомним, что адрес объекта данных возвращается функцией LOC.

```
!ms$pack: 1
type pair1
 integer(1) a
 integer(4) b
end type pair1
type (pair1) :: x1 = pair1(3, 3)
!ms$pack: 2
type pair2
 integer(1) a
 integer(4) b
end type pair2
type (pair2) :: x2 = pair2(3, 3)
!ms$pack: 4
type pair4
 integer(1) a
```

```
 integer(4) b
end type pair4
type (pair4) :: x4= pair4(3, 3)
!ms$pack:
type pair8                          ! Устанавливаемое по умолчанию
 integer(1) a                       ! выравнивание равно 8 байтам
 integer(4) b
end type pair8
type (pair8) :: x8= pair8(3, 3)
print *, loc(x1.b) - loc(x1.a)      !      1
print *, loc(x2.b) - loc(x2.a)      !      2
print *, loc(x4.b) - loc(x4.a)      !      4
print *, loc(x8.b) - loc(x8.a)      !      4
end
```

П.1.10. Метакоманда $ATTRIBUTES

Метакоманда $ATTRIBUTES объявляет Microsoft-атрибуты процедур и переменных. Эти атрибуты являются расширением FPS4 по отношению к предусмотренным стандартом Фортран 90 атрибутам. Синтаксис:

$ATTRIBUTES *attribute-list* :: *variable-list*

или

!MS$ATTRIBUTES attribute-list :: variable-list

attribute-list - один или более объявляемых для переменной атрибутов Microsoft. При объявлении более одного атрибута последние разделяются запятыми.

variable-list - список переменных или процедур. При наличии в списке более одного элемента последние разделяются запятыми.

Атрибуты Microsoft упрощают передачу данных и процедур между Фортраном и другим языком программирования, например СИ. Объявление атрибутов должно появляться в области объявления данных программной единицы. Атрибуты Microsoft:

ALIAS	EXTERN	VARYING	C
REFERENCE	DLLEXPORT	STDCALL	DLLIMPORT
VALUE			

Пример.

```
interface                           ! Те же атрибуты должна содержать
  subroutine for_sub (i)            ! и подпрограмма for_sub
    !ms$attributes c, alias:'_for_sub' :: for_sub
    integer i
  end subroutine for_sub
end interface
```

Подробное рассмотрение атрибутов Microsoft дано в прил. 2.

П.1.11. Метакоманды и опции компилятора

Некоторые метакоманды и опции компилятора оказывают одинаковый эффект на компилятор. Список таких метакоманд приведен в табл. П.1.1.

Таблица П.1.1. Метакоманды и опции компилятора

Метакоманда	*Эквивалентная опция компилятора*
$DEBUG	/4Yb
$NODEBUG	/4Nb
$DECLARE	/4Yd
$NODECLARE	/4Nd
$DEFINE symbol	/D*symbol*
$INTEGER:option	/4I*option*
$FIXEDFORMLINESIZE:option	/4L*option*
$FREEFORM	/4Yf
$NOFREEFORM	/4Nf
$OPTIMIZE: 'ON:option'	/O*option*
$PACK:option	/Zp*option*
$REAL:option	/4R*option*
$STRICT	/4Ys
$NOSTRICT	/4Ns

Метакоманда в отличие от опции компилятора может оказывать действие на часть программы и при необходимости может быть отключена. Опция компилятора действует во время всего процесса компиляции, если только ее действие не прервано или изменено расположенной в исходном коде метакомандой.

Замечание. Любая метакоманда может начинаться с префикса !MS$.

Приложение 2. Microsoft-атрибуты

Microsoft-атрибуты являются расширением FPS по отношению к стандарту Фортран 90. Эти атрибуты придают объектам FPS (переменным и процедурам) дополнительные свойства, позволяющие соединять написанные на разных языках программы, например на Фортране и СИ, и использовать хранящиеся в динамических библиотеках процедуры. Программирование на разных языках в ряде случаев имеет преимущества, поскольку позволяет:

- использовать уже существующий написанный на другом языке код;
- разрабатывать процедуры на другом языке, трудновыполнимые на основном;
- увеличить скорость выполнения программы.

Microsoft поддерживает 32-разрядные версии Фортрана, Visual C++, Visual Basic и MASM; смешанное программирование возможно с применением любого из перечисленных языков.

Смешанное программирование требует соблюдения правил присваивания имен переменным и процедурам, использования стека и передачи параметров между написанными на разных языках процедурами. Совокупность этих правил называется *соглашениями вызова.*

Соглашения вызова включают:

- стек:

  получает ли процедура переменное или фиксированное число параметров?

  какая процедура очищает стек после вызова?

- соглашение по именам:

  значим ли регистр в имени?

  имеет ли имя ограничения по форме (как в Visual C++)?

- протокол передачи параметров:

  передаются ли параметры по значению или по ссылке?

  какие типы данных являются эквивалентными в различных языках?

Перечень Microsoft-атрибутов и объекты, к которым они могут быть применены, дан в табл. П.2.1.

Таблица П.2.1. Microsoft-атрибуты

Атрибут	*Объявление переменных*	*Оператор COMMON*	*Операторы определения процедур и оператор EXTERNAL*
ALIAS	Да	Да	Да
C	“	“	“
DLLEXPORT	“	“	“
DLLIMPORT	“	“	“

EXTERN	Да	Нет	Нет
REFERENCE	"	"	"
STDCALL	"	Да	Да
VALUE	"	Нет	Нет
VARYING	Нет	"	Да

В FPS1 Microsoft-атрибут объекта указывался в квадратных скобках сразу после имени объекта. Если объект имел более одного атрибута, то они разделялись запятыми. В FPS4 Microsoft-атрибуты вводятся метакомандой !MS$ATTRIBUTES.

Microsoft-атрибуты могут быть использованы при определении процедур и при объявлении типов данных с операторами INTERFACE и ENTRY. Эти атрибуты могут быть переданы в программную единицу через *use*-ассоциирование.

П.2.1. Атрибут ALIAS

Атрибут задает внешнее имя подпрограммы, функции или общего блока данных. Синтаксис:

ALIAS : *внешнее имя*

внешнее имя - символьная константа (заключенная в кавычки последовательность символов) (может быть СИ-строкой). В отличие от имен Фортрана в имени ALIAS является значащим регистр букв. Так, ALIAS : 'PAT' и ALIAS : 'Pat' - разные имена. Для *внешнего имени* могут быть использованы недопустимые в Фортране, но приемлемые для другого языка имена, например имена, начинающиеся с символа подчеркивания.

В исходном файле вызов процедуры может быть выполнен только по ее основному имени. Вне исходного файла процедура может быть вызвана только по ее псевдониму. Атрибут ALIAS используется в операторе INTERFACE и в самой процедуре для переопределения ее имени.

Атрибут ALIAS имеет больший приоритет, чем атрибут C. Если атрибут ALIAS использован для процедуры совместно с атрибутом C, то при вызове процедуры будут использованы применяемые в СИ соглашения. Внешнее имя процедуры по-прежнему будет определяться атрибутом ALIAS.

Атрибут не может быть использован с внутренними процедурами и формальными параметрами.

Пример. В СИ подпрограмма *iname* будет вызываться по имени *_ename*.

```
subroutine iname
 !ms$attributes c, alias: '_ename' :: iname
 print *, 'Called'
end subroutine

 call iname()                    ! В Фортране вызов по-прежнему
end                              ! выполняется по имени iname
```

П.2.2. Атрибуты C и STDCALL

При использовании функций СИ или процедур ассемблера совместно с Фортран-программами необходимо указать, каким образом выполняется передача данных. FPS поддерживает два соглашения о вызовах, задаваемых атрибутами C и STDCALL. Эти атрибуты применимы к процедурам, общим блокам и типам данных. При использовании с процедурой атрибуты определяют набор соглашений о вызовах. Различия между используемыми соглашениями приведены в табл. П.2.2.

Таблица П.2.2. Соглашения о вызовах

Соглашение	*C*	*STDCALL*	*По умолчанию*
Добавление ведущего подчеркивания	Да	Да	Да
Добавление числа параметров	Нет	"	"
Чувствительность к регистру букв	Да	"	Нет
Обработка параметров справа налево	"	"	Да
Очистка стека	"	Нет	Нет
Переменное число параметров	"	"	"
Передача длины для строки	Нет	"	Да
Передача параметров по значению[1]	Да	Да	Нет

Задаваемые по умолчанию C и STDCALL соглашения добавляют код в вызываемую процедуру, который восстанавливает стек, когда завершаются вычисления в вызванной процедуре. Это обеспечивает генерацию меньших по размеру программ, по сравнению с C-соглашениями, при которых такой код следует за каждым вызовом процедуры.

По умолчанию параметры процедур, определенных с атрибутами C и STDCALL передаются по значению. Однако их можно передать и по ссылке, применив атрибут REFERENCE. В процедуры, использующие стандартные соглашения Фортрана о вызовах, аргументы передаются по ссылке (при условии, что для них не указан атрибут VALUE).

Имена, объявленные в процедурах, использующих атрибут C, автоматически модифицируются к виду, применяемому при C-вызовах. Внешние имена переводятся на нижний регистр и предваряются знаком подчеркивания (_). Для использования имен, содержащих прописные буквы, необходимо применить атрибут ALIAS.

Внешние имена в процедурах, использующих атрибут STDCALL, переводятся на нижний регистр, предваряются знаком подчеркивания и завершаются знаком @ и числом передаваемых параметрами байт. Например, подпрограмма с именем PROC, имеющая 4 параметра типа INTEGER(4) и определенная с атрибутом STDCALL, получит внешнее имя *_proc*@16.

---

[1] Имена массивов и строк в СИ являются указателями, что обеспечивает их передачу в СИ по ссылке.

Атрибуты не могут быть применены к формальным параметрам за исключением параметров типа INTEGER.

П.2.3. Атрибут EXTERN

Атрибут используется при объявлении глобальных переменных и указывает, что переменная объявлена в другом файле, написанном на отличном от Фортрана языке программирования. Атрибут не может быть использован для формальных параметров процедур.

П.2.4. Атрибут REFERENCE

Атрибут используется только для формальных параметров и обеспечивает передачу параметров *по ссылке*, а не по значению. При передаче по ссылке формальный параметр адресует фактический параметр, а не его копию. Поэтому изменение в процедуре значения формального параметра приведет к изменению и значения фактического параметра. По умолчанию все параметры Фортрана передаются по ссылке.

П.2.5. Атрибут VALUE

Атрибут обеспечивает передачу параметров *по значению*. В этом случае формальный параметр получает копию фактического. Фактический параметр при передаче по значению останется без изменений, даже если в процедуре изменено значение соответствующего формального параметра. Это обусловлено тем, что формальный параметр адресует не фактический параметр, а его копию.

Когда формальный параметр имеет атрибут VALUE, то передающий ему значение фактический параметр может быть другого типа. Если необходимо преобразование типа, то оно выполняется до вызова.

Если при определении процедуры задан атрибут C, то все параметры принимают атрибут VALUE, поскольку в СИ по умолчанию параметры передаются по значению. Правда, в этом случае строки, подстроки, перенимающие размер и форму массивы не могут быть переданы по значению (поскольку в СИ подобные объекты являются указателями и передаются по ссылке).

Если нужно передать символьные данные и перенимающие размер и форму массивы по значению, необходимо использовать атрибут VALUE.

Чтобы обрабатывать переданный массив не как структуру, а как массив, необходимо выполнить одно из двух действий:

- использовать для формального параметра атрибут REFERENCE;
- передать адрес, возвращаемый функцией LOC, по значению.

Пример.

```
Integer  kr /2/, kv /2/
write(*, *) kr, kv                !    2      2
call refval(kr, kv)
write(*, *) kr, kv                !    4      2
end
```

```
subroutine refval(kr, kv)
 !ms$ attributes reference :: kr    ! Параметр передается по ссылке
 !ms$ attributes value :: kv        ! Параметр передается по значению
 integer kr, kv
 kr = kr + 2
 kv = kv + 2                        ! Измененное значение формального
end                                 ! параметра не передается фактическому
```

П.2.6. Атрибут VARYING

Атрибут используется только для подпрограмм и функций, объявленных с атрибутом C. Задание атрибута VARYING позволяет обращаться к процедуре с разным число параметров. При этом числом фактических (передаваемых) параметров не должно превышать число формальных параметров.

Поскольку вызов с разным числом параметров предусмотрен стандартом Фортран 90, то атрибут VARYING может использоваться для вызова процедуры из программы, написанной на другом языке, например СИ. В FPS процедура с таким атрибутом может быть вызвана с переменным числом параметров, если программа компилируется с опцией компилятора /4fps1. Форма списков аргументов для атрибута VARYING несовместима с формой, используемой в Фортране 90.

В FPS1 процедура должна сама определять число полученных параметров. Для этого в список параметров вводится параметр, в котором указывается количество передаваемых параметров. Использование ключевых слов не поддерживается. Фактические и формальные параметры должны быть совместимы по порядку следования и типам.

Пример.

```
interface to subroutine varpar [varying, c] (k, a, n)
  integer k, a(*), n
end
integer a(10) /1, 2, 3, 4, 5, 6, 7, 8, 9, 10/
call varpar(2, a, 6)                ! Передаем 2 параметра
call varpar(1, a)                   ! Передаем 1 параметр
end

subroutine varpar (k, a, n)         ! Атрибуты указаны в операторе interface to
 integer a(*), k, n, i
 if(k .eq. 1) n = 3
 write(*, *) (a(i), i = 1, n)
end
```

Замечание. В примере при определении атрибутов использован синтаксис FPS1. При компиляции применяется опция компилятора /4fps1.

П.2.7. Атрибуты DLLEXPORT и DLLIMPORT

Атрибуты DLLEXPORT и DLLIMPORT задают интерфейс к динамически подключаемым библиотекам. Также эти атрибуты могут быть применены к данным.

DLLEXPORT объявляет, что функция или данные экспортируются в другие приложения или динамические библиотеки DLL. При использовании этого атрибута компьютер создает наиболее эффективный код, и нет необходимости задавать файл определения модуля (DEF-файл) для экспорта символов. Задание атрибута DLLEXPORT должно выполняться в той же программной единице, где объявляются функции или данные, к которым атрибут применяется.

Программа, использующая определенные в DLL символы, импортирует их. В такой программе должен быть объявлен атрибут DLLIMPORT. Объявление выполняется в той программной единице, которая импортирует символы. Атрибут DLLIMPORT размещается в разделе объявлений программной единицы.

Пример.

```
subroutine arraytest(arr)
 !ms$attributes dllexport :: arraytest
 real arr(3, 7)
 integer i, j
 do i = 1, 3
  do j = 1, 7
   arr (i, j) = 11.0 * i + j
  enddo
 enddo
end subroutine
```

Чтобы включить подпрограмму в динамическую библиотеку, необходимо создать DLL-проект. Для этого используется цепочка File - New - Project Workspace - Dynamic-Link-Library - внести имя проекта в поле Name - Create. Пусть имя проекта FDLL. Затем, применяя цепочку Insert - Files Into Project, добавить в проект файл, например arr.f90, содержащий текст подпрограммы *arraytest*. Таким же образом в проект добавляются и другие исходные файлы. После этого создать DLL-библиотеку: Build - Build fdll.dll. Пусть файлы библиотеки fdll.lib и fdll.dll размещены в папке FDLL. Для использования динамической библиотеки fdll.dll в программе необходимо до генерации содержащего эту программу проекта добавить в опции компоновщика строку с именем библиотеки fdll.lib. Это можно сделать так: Build - Settings - Link - выбрать категорию General - добавить в поле Object/Library modules после библиотеки kernel32.lib строку "c:\fdll\fdll.lib". (Разделителем между именами библиотек является пробел.) Затем сгенерировать проект. Пусть результатом построения проекта является файл ted.exe. При запуске программы ted.exe, использующей библиотеку fdll.dll, ей должен быть известен путь к файлу fdll.dll. Например, файл fdll.dll можно разместить там же, где находится файл ted.exe.

```
program ted                          ! Определяем имя импортируемого символа
!ms $attributes dllimport :: arraytest
 integer i, j
 real arr(3, 7)
 call arraytest(arr)                 ! Подпрограмма будет взята из fdll.dll
 print '(1x, 7f5.1)', ((arr(i, j), j = 1, 7), i = 1, 3)
end program
```

Приложение 3. Дополнительные процедуры FPS

FPS, помимо предусмотренных стандартом Фортран 90 процедур, содержит большое число дополнительных процедур, котрые в документации FPS распределены по разделам:

Раздел	*Модуль*
Запуск программ	MSFLIB
Управление программой	MSFLIB
Работа с системой, дисками и директориями	MSFLIB
Управление файлами	MSFLIB
Случайные числа	MSFLIB
Процедуры даты и времени	MSFLIB
Процедуры клавиатуры и звука	MSFLIB
Обработка ошибок	MSFLIB
Аргументы в командной строке	MSFLIB
Сортировка и поиск в массиве	MSFLIB
Процедуры управления операциями с плавающей точкой	MSFLIB
Процедуры QuickWin	MSFLIB
Графические процедуры	MSFLIB
Создание диалогов	DIALOGM
Работа с национальным языком	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

Функция RUNQQ используется для запуска другого процесса.

Процедура	*Тип*	*Назначение*
RUNQQ	Функция	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	*Тип*	*Назначение*
RAISEQQ	Функция	Передает сигнал прерывания выполняемой программе, моделирующий прерывание операционной системы
SIGNALQQ	"	Управляет обработкой сигналов
CALL SLEEPQQ	Подпрограмма	Задерживает исполнение программы на указанный в миллисекундах промежуток времени

П.3.3. Работа с системой, дисками и директориями

Процедуры используются для работы с физическими устройствами, директориями и для идентификации полных имен путей.

Процедура	*Тип*	*Назначение*
CHANGEDIRQQ	Функция	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	Подпрограмма	Добавляет новую переменную окружения или устанавливает значение существующей переменной
SYSTEMQQ	Функция	Выполняет команду путем передачи командной строки интерпретатору команд операционной системы

П.3.4. Управление файлами

Процедуры используются для управления файлами и получения информации, хранящейся о файлах в операционной системе.

Процедура	*Тип*	*Назначение*
DELFILESQQ	Функция	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	Подпрограмма	Упаковывает время для использования функцией SETFILETIMEQQ
RENAMEFILEQQ	Функция	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 компоненты
CALL UNPACKTIMEQQ	Подпрограмма	Распаковывает упакованное время и дату в отдельные компоненты

П.3.5. Генерация случайных чисел

Помимо предусмотренных стандартом подпрограмм

CALL RANDOM_NUMBER

и

CALL RANDOM_SEED

для получения случайных чисел можно использовать следующие подпрограммы:

Процедура	*Тип*	*Назначение*
CALL RANDOM	Подпрограмма	Возвращает псевдослучайное вещественное число, большее или равное нулю и меньшее единицы
CALL SEED	"	Изменяет начальную точку генератора псевдослучайных чисел

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	"	Возвращает системное время
SETDAT	Функция	Установка даты
SETTIM	"	Установка времени

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	*Тип*	*Назначение*
CALL BEEPQQ	Подпрограмма	Воспроизводит сигнал с заданными в миллисекундах продолжительностью и частотой
GETCHARQQ	Функция	Возвращает введенный символ
GETSTRQQ	"	Читает символьную строку с клавиатуры, используя буфер
PEEKCHARQQ	"	Проверяет, была ли нажата какая-либо клавиша консоли

П.3.8. Обработка ошибок

Процедуры управляют обработкой критических, приводящих к завершению программы ошибок, и позволяют получить дополнительную информацию о причинах иных ошибок.

Процедура	*Тип*	*Назначение*
GETLASTERRORQQ	Функция	Возвращает последнюю обнаруженную дополнительной процедурой ошибку
CALL MATHERRQQ	Подпрограмма	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

Приложение 4. Нерекомендуемые и устаревшие 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.4.1. Нерекомендуемые свойства Фортрана

П.4.1.1. Фиксированный формат записи исходного кода

По умолчанию длина строки исходного текста при записи его в фиксированном формате равна 72 символам. Однако в результате применения метакоманды $FIXEDFORMLINESIZE (прил. 1) она может быть увеличена до 80 или 132 символов.

Интерпретация символов строки Фортран-программы в фиксированном формате зависит от того, в какой колонке они указаны. Правила интерпретации сведены в табл. П.4.1.

Таблица П.4.1. Интерпретация символов строки программы

Колонки	*Интерпретация символа*
1	Символы $ или !MS$ (указывает на метакоманду)
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 фиксированном формате различают 5 типов строк: *комментарии, начальные строки, строки продолжения, метакоманды* 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.

Метакоманды (прил. 1) управляют работой компилятора. Метакоманда начинается с символа $ или префикса !MS$. В любом случае при фиксированном формате метакоманда должна начинаться в первой колонке строки, например:

```
$NOFREEFORM
!MS$NOFREEFORM
```

Отладочные строки. Включение режима проверки осуществляется метакомандой $DEBUG. Выключение - метакомандой $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 продожения.

```
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.4.1.2. 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.4.1.

```
:01:02:03:04:05:06:07:
:-----a-----:
            :-----b-----:
:--c(1)--:--c(2)--:
```

Рис. П.4.1. 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* к 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 метакоманда $STRICT или опция компилятора /4Ys, то при работе с оператором EQUIVALENCE следует придерживаться 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 из-

менено 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.4.1.3. 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.4.1.4. 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.4.1.5. 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.4.2. Устаревшие свойства Фортрана

П.4.2.1. 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
enddo
print *, k1, k2, k3                !      2       3       5
end
```

Оператор заменяется конструкциями IF и SELECT CASE.

П.4.2.2. 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. 9.4). Назначаемый GOTO всегда может быть успешно заменен конструкциями IF или SELECT CASE.

П.4.2.3. 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
endif
goto vi (100, 200, 400)
100 continue
. . .
goto 500
200 continue
. . .
goto 500
400 continue
. . .
500 continue
end
```

П.4.2.4. 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
  . . .
enddo
```

Для приведенного цикла естественно ожидать, что число итераций *ni*, которое вычисляется по формуле

$$ni = max(int(stop - start + inc)/inc), 0),$$

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.

Потеря точности может произойти и в длинных циклах, например, при многократном сложении чисел.

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.4.2.5. 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.4.2.6. 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.4.2.7. 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.

Литература

1. Бродин В. Б., Шагурин И. И. Микропроцессор i 486. Архитектура, программирование, интерфейс. - М.: ДИАЛОГ-МИФИ, 1993. - 240 с.
2. Вельбицкий И. В. Технология программирования. - Киев: Техника, 1984. - 279 с.
3. Корриган Дж. Компьютерная графика: Секреты и решения. - М.: Энтроп, 1995. - 352 с.
4. Лингер Р., Миллс Х., Уитт Б. Теория и практика структурного программирования. - М.: Мир, 1982. - 408 с.
5. Майерс Г. Искусство тестирования программ. - М.: Финансы и статистика, 1982. - 176 с.
6. Меткалф М., Рид. Дж. Описание языка программирования Фортран 90. - М.: Мир, 1995. - 302 с.
7. Любимский Э. З., Мартынюк В. В., Трифонов Н. П. Программирование. - М.: Наука, 1980. - 608 с.
8. Першиков В. И., Савинков В. М. Толковый словарь по информатике. - М.: Финансы и статистика, 1991. - 543 с.
9. Простяков И. Н. Руководство по компиляции и линкованию на Microsoft Фортране 5.0. Черкасск: A.B.C., 1991. - 42 с.
10. Скляров В. А. Язык C++ и объекто-ориентированное программирование. - Мн.: Высш. шк., 1997. - 478 с.
11. Соловьев П. В. Фортран для персонального компьютера. - М.: Арист, 1991. - 223 с.
12. Холстед М. Начала науки о программах. - М.: Финансы и статистика, 1981. - 128 с.
13. Шикин Е. В., Боресков А. В. Компьютерная графика. Динамика, реалистические изображения. - М.: ДИАЛОГ-МИФИ, 1995. - 288 с.
14. Эйнарссон Бо, Шокин Ю. И. Фортран 90. Книга для программирующих на языке Фортран 77. - Новосибирск: Изд-во СО РАН "Инфолио", 1995. - 185 с.
15. Эндерле Г., Кэнси К., Пфафф Г. Программные средства машинной графики. Международный стандарт GKS. - М.: Радио и связь, 1988. - 480 с.